High Performance Computing via High Level Synthesis

ScuDo
Scuola di Dottorato ~ Doctoral School
WHAT YOU ARE, TAKES YOU FAR

Doctoral Dissertation
Doctoral Program in Electronic Engineering (30^{th}cycle)

High Performance Computing via High Level Synthesis
Xilinx FPGA

By

Mehdi Roozmeh

******

Supervisor(s):
Prof. Luciano Lavagno

Doctoral Examination Committee:
Prof. Roberto Passerone , Referee, University of Trento
Prof. Davide Quaglia, Referee, University of Verona
Prof. E.F, University of...
Prof. G.H, University of...
Prof. I.J, University of...

Politecnico di Torino
2018

Declaration

I hereby declare that, the contents and organization of this dissertation constitute my own original work and does not compromise in any way the rights of third parties, including those relating to the security of personal data.

Mehdi Roozmeh
2018

* This dissertation is presented in partial fulfillment of the requirements for **Ph.D. degree** in the Graduate School of Politecnico di Torino (ScuDo).

I would like to dedicate this thesis to my loving parents

Acknowledgements

First of all I appreciate the given opportunity to study and educate in Politecnico di Torino, during my PhD course in the Department of Electronics and Telecommunications (DET) I met great and outstanding people which is impossible to name all, but I offer my sincerest gratitude to all who helped me to complete my study and find my future path.

Secondly, I would like to express my sincere gratitude to my advisor Prof. Luciano Lavagno for the continuous support of my Ph.D study and related research, for his patience, motivation, and immense knowledge. His guidance helped me in all the time of research and writing of this thesis. I could not have imagined having a better advisor and mentor for my PhD study.

Last but not least, my love goes to my parents and sister for the inspiration I draw from them during my life.

Mehdi Roozmeh
2018, Torino

Abstract

As more and more powerful integrated circuits are appearing on the market, more and more applications, with very different requirements and workloads, are making use of the available computing power. This thesis is in particular devoted to High-Performance Computing applications, where those trends are carried to the extreme. In this domain, the primary aspects to be taken into consideration are (1) performance (by definition) and (2) energy consumption (since operational costs dominate over procurement costs). These requirements can be satisfied more easily by deploying heterogeneous platforms, which include CPUs, GPUs and FPGAs to provide a broad range of performance and energy-per-operation choices. In particular, as we will see, FPGAs clearly dominate both CPUs and GPUs in terms of energy, and can provide comparable performance.

An important aspect of this trend is of course design technology, because these applications were traditionally programmed in high-level languages, while FPGAs required low-level RTL design. The OpenCL(Open Computing Language) developed by the Khronos group enables developers to program CPU, GPU and recently FPGAs using functionally portable (but sadly not performance portable) source code which creates new possibilities and challenges both for research and industry [1].

FPGAs have been always used for mid-size designs and ASIC prototyping thanks to their energy efficient and flexible hardware architecture, but their usage requires hardware design knowledge and laborious design cycles. Several approaches are developed and deployed to address this issue and shorten the gap between software and hardware in FPGA design flow, in order to enable FPGAs to capture a larger portion of the hardware acceleration market in datacenters. Moreover, FPGAs usage in data-centers is growing already, regardless of and in addition to their use as

[1] Both Intel and Xilinx provide software development kit(SDK) to support high level synthesis using the OpenCL programming language

computational accelerators, because they can be used as high performance, low power and secure switches inside data-centers.

High-Level Synthesis (HLS) is the methodology that enables designers to map their applications on FPGAs (and ASICs). It synthesizes parallel hardware from a model originally written C-based programming languages .e.g. C/C++, SystemC and OpenCL. Design space exploration of the variety of implementations that can be obtained from this C model is possible through wide range of optimization techniques and directives, e.g. to pipeline loops and partition memories into multiple banks, which guide RTL generation toward application dependent hardware and benefit designers from flexible parallel architecture of FPGAs.

Model Based Design (MBD) is a high-level and visual process used to generate implementations that solve mathematical problems through a verified set of IP-blocks. MBD enables developers with different expertise, e.g. control theory, embedded software development, and hardware design to share a common design framework and contribute to a shared design using the same tool. Simulink, developed by Matlab, is a model based design tool for simulation and development of complex dynamical systems. Moreover, Simulink embedded code generators can produce verified C/C++ and HDL code from the graphical model. This code can be used to program micro-controllers and FPGAs. This PhD thesis work presents a study using automatic code generator of Simulink to target Xilinx FPGAs using both HDL and C/C++ code to demonstrate capabilities and challenges of high-level synthesis process. To do so, firstly, digital signal processing unit of a real-time radar application is developed using Simulink blocks. Secondly, generated C based model was used for high level synthesis process and finally the implementation cost of HLS is compared to traditional HDL synthesis using Xilinx tool chain.

Alternative to model based design approach, this work also presents an analysis on FPGA programming via high-level synthesis techniques for computationally intensive algorithms and demonstrates the importance of HLS by comparing performance-per-watt of GPUs(NVIDIA) and FPGAs(Xilinx) manufactured in the same node running standard OpenCL benchmarks. We conclude that generation of high quality RTL from OpenCL model requires stronger hardware background with respect to the MBD approach, however, the availability of a fast and broad design space exploration ability and portability of the OpenCL code, e.g. to CPUs and GPUs, motivates FPGA industry leaders to provide users with OpenCL software

development environment which promises FPGA programming in CPU/GPU-like fashion. Our experiments, through extensive design space exploration(DSE), suggest that FPGAs have higher performance-per-watt with respect to two high-end GPUs manufactured in the same technology(28 nm). Moreover, FPGAs with more available resources and using a more modern process (20 nm) can outperform the tested GPUs while consuming much less power at the cost of more expensive devices.

Contents

List of Figures

List of Tables

Chapter 1

FPGA - Field Programmable Gate Arrays

Field programmable arrays(FPGA) consists of arrays of gates that can be programmed and reconfigured by a designer. Hardware description languages (VHDL/Verilog) are widely used to model a design in a bit and cycle accurate way and map them to FPGA via various available synthesis tools to generate functionally verified bit-stream in an automated flow[1, 2]. Although, multiple companies share FPGA's market, but Intel(after acquisition of Altera in 2016) and Xilinx are two major competitors. Even though Intel cannot be underestimated especially when it comes to their technology and capital, this work focuses on Xilinx tools and technology. Xilinx is the market leaders of FPGAs with 18-months technology lead. Its products are aimed to meet requirements of various workloads coming from different domains[3, 4], figure 1.2 suggests that more than 50% of FPGA market belongs to Xilinx programmable platforms.

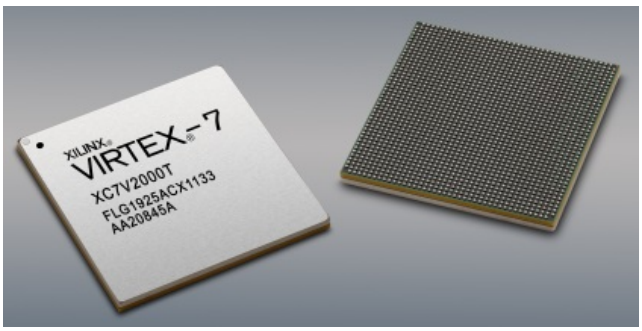


Figure 1.1 28 nm Xilinx Virtex7 FPGA

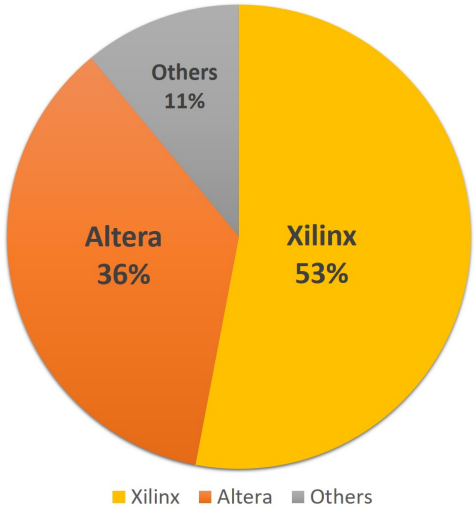


Figure 1.2 Major share holders of FPGA market

As technology world is shifting toward a huge ramp driven by 5G communication, Artificial intelligence(AI) and Internet of Things(IoT), the demand of high performance computing and efficient-energy solutions are growing and this makes FPGA more suitable to be used as the core of embedded electronic systems. However, complexity of FPGA is growing day by day, but FPGAs development environment improvement in recent years made developers to consider them as a cost optimized and high performance devices, Figures 1.3 and 1.4 demonstrate increasing growth of FPGA deployment in USA and Asian pacific markets, overall annual market value of FPGA is estimated around 7 billion USD by the end of 2022 [5–7].

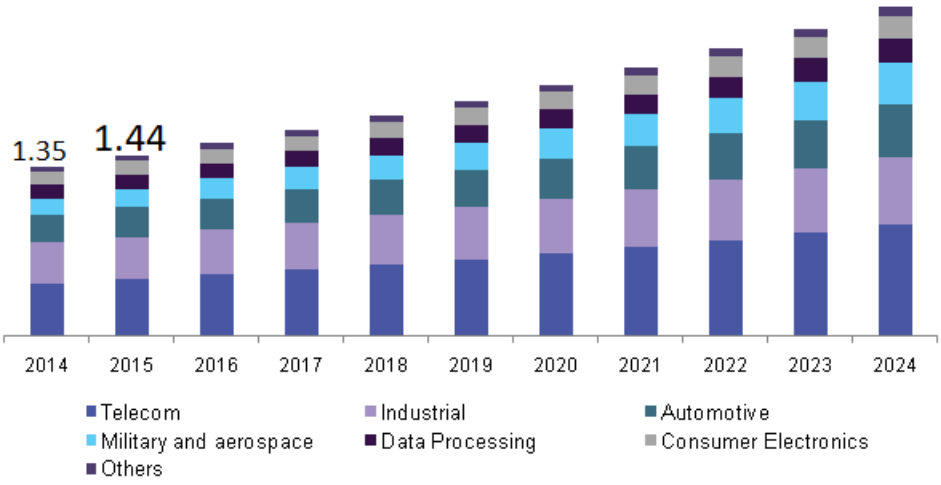


Figure 1.3 U.S. FPGA Market by application, 2014-2024(USD Bilion)

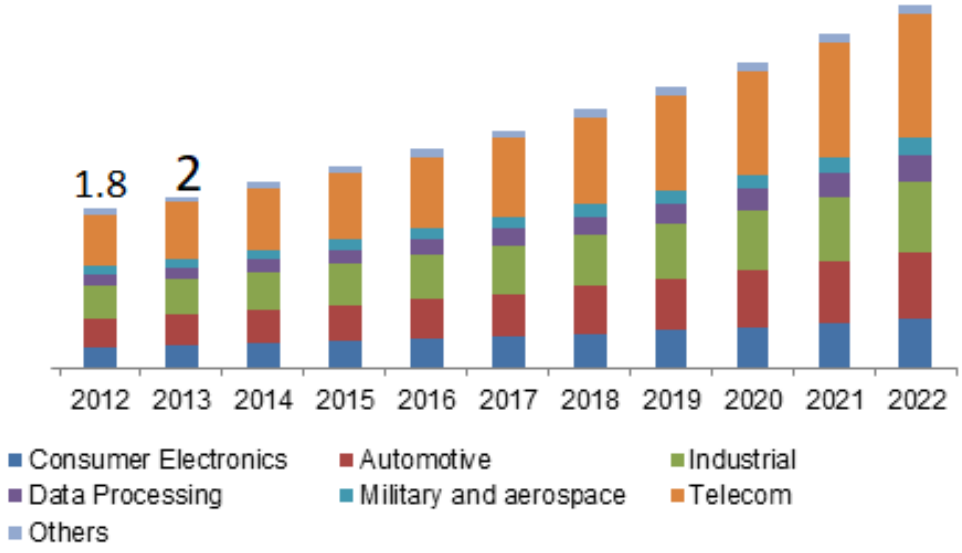


Figure 1.4 Asian Pacific FPGA Market by application, 2014-2024 (USD Bilion)

Moreover, previously most of the traffics in the INTERNET were between client and data-centers which is known as north-south traffic in data-center terminology, nowadays, 80 percent of the total traffic belongs to east-west traffic which is the internal communications within a data-center [8], figure 1.5 shows top view of data center architecture. This fact increases FPGA deployment in data centers mainly because of the FPGAs flexibility that can be configured as high-performance switches inside data centers structures without outside standards concern coming from client side[9].

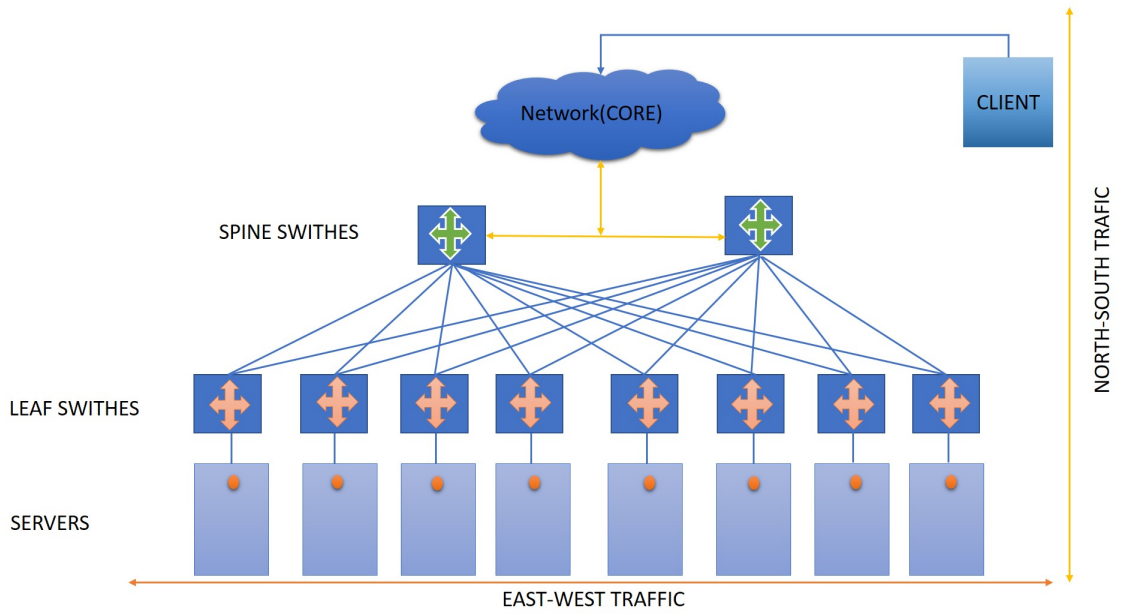


Figure 1.5 Top view of Data center

In the rest of this chapter, FPGAs are discussed both from hardware and software perspective which are fundamentals to understand and perform successful software and hardware co-design targeting FPGAs.

1.1 An example of a modern FPGA-based platform

FPGAs, like any other integrated circuit, are mounted on boards (and often racks of boards) to create a complete electronic system. To give an idea of the power and possibilities of modern FPGA-based platforms, we will consider a very recent example board, shown in figure 1.6. HTG-9200 from HiTech Global is powered by the latest generation Xilinx Virtex Ultrascale+(16nm) FPGA. This modern platform is suitable, for example, for high performance optical networking applications since it has nine QSFP28[1] ports and two separate DDR4 high-performance DRAM banks(using a total of 10 memory chips).

[1]Quad Small Form-Factor Pluggable is a high-performance and low-power Ethernet connectivity solution for data center and high performance computing. These transceivers come in various types supporting 40 G and 100 G Ethernet.

The HTG9200 is considerably more powerful than most FPGA boards that are deployed today. However, it is a clear example of an ongoing trend to replace ASICs with FPGAs for any application domain that require the latest technologies, but cannot afford to design an ASIC due to the Non-Recurrent Engineering cost. Design costs for the 14 nm technology node are estimated by various industry sources to be around 300M$ per design, and mask costs are most likely comparable. This means that several million integrated cicuits must be sold to recover the NRE costs. For this reason, and due to the ability to upgrade the hardware in situ, FPGAs are becoming the preferred platform for a growing number of applications, ranging from automotive (ADAS in particular), to telecom, military, aerospace, and recently even data centers. In the latter case, FPGAs are not competing with ASICs but with GPUs, and can offer comparable performance at a fraction of the energy-per computation cost, as we will show later in this thesis.

As we discussed, a key motivation of FPGAs is to keep design costs low. This is achieved first and foremost by reducing the verification costs dramatically, since their reprogrammability ensures that there is no need to get the first implementation completely bug-free. Designs can be reloaded at will, and debugging on the FPGA itself is much easier than on an ASIC.

Moreover, recent design technology advances, in particular High-Level Synthesis, have enabled a dramatic paradigm shift in terms of how FPGAs (and ASICs) are designed. As we will demonstrate in this thesis, HLS tools enable the designer to achieve high-quality implementations in a fraction of the time required by traditional HDL-based methodologies working at the Register Transfer level. The Quality of Results of RTL can even be surpassed by HLS-based solutions, thanks to the broader design space (e.g. the tradeoffs between pipelining and resource consumption, or the exploration of memory architectures) that is afforded by HLS. This motivates us to conduct an extensive research on different FPGA programming approaches and report our experiment results in this dissertation.

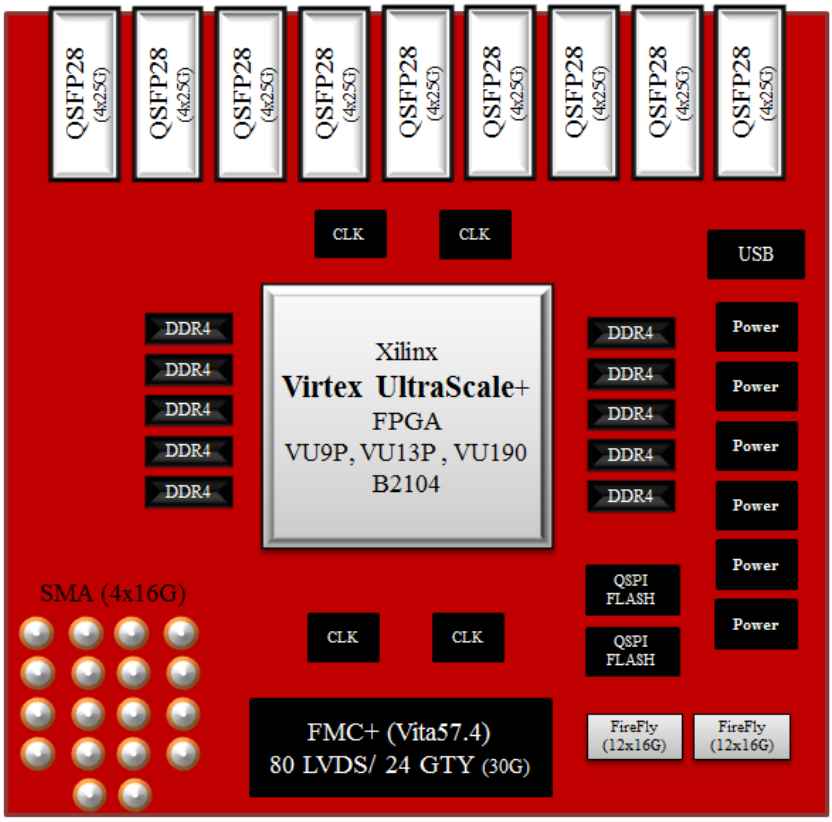


Figure 1.6 Xilinx Ultrascale+ Platform Scheme

1.2 FPGA Architecture

FPGAs are capable to serve as an accelerator for the wide range of applications ,moreover, new technologies shape and develop FPGA platforms day by day to meet customer and market needs. Modern FPGAs consist of millions of logic cells and switches which are programmable. This chapter studies the key components of FPGA chip and provide the numerical report of available resources for each component in 7 series Xilinx FPGAs(28 nm).

1.2.1 Configurable Logic Blocks

Figure 1.7 presents top view of FPGA fabric composed of arrays of configurable logic blocks(CLB) and switches.

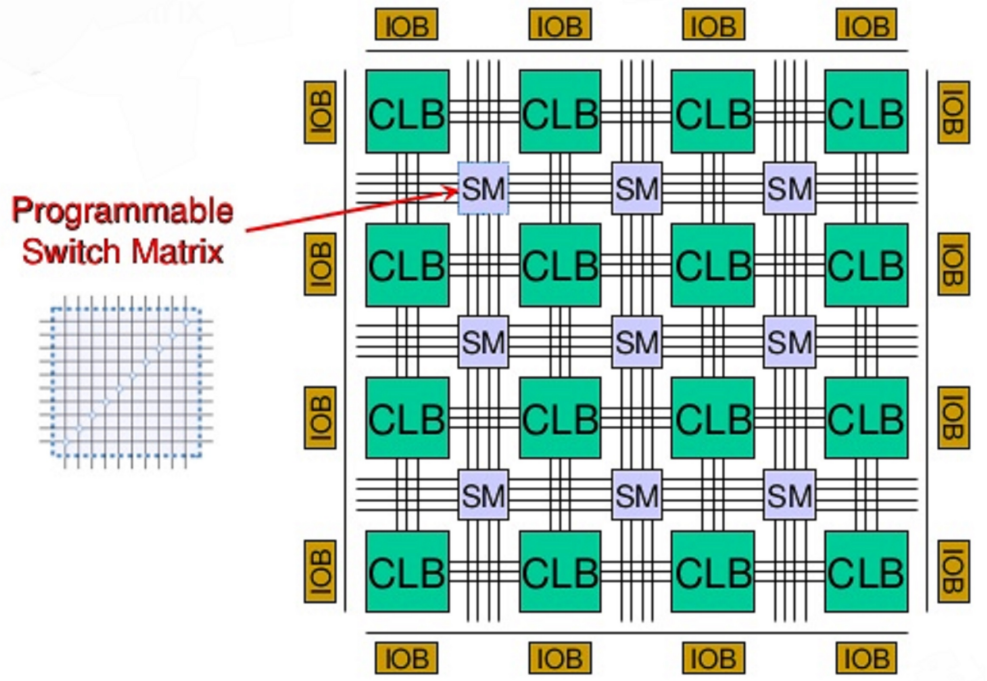


Figure 1.7 Top View of FPGA fabric architecture

CLBs are the key component in Xilinx FPGAs and each has pair of slices, figure 1.8 shows simplified graphical presentation of one slice with 6-input look-up tables(LUT), fast adders and registers. Logical,arithmetic,memory and shift register functions can be implemented using these slices [10].

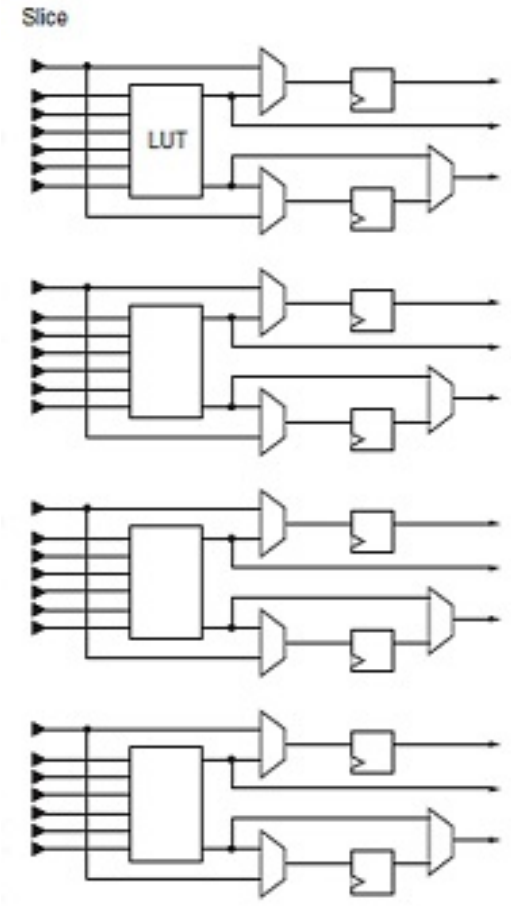


Figure 1.8 Slice of CLB

1.2.2 DSP Slice

Xilinx FPGAs provide slices of DSP that are designed to implement vast number of binary multipliers and adders used in DSP applications. Figure 1.9 provides insight to DSP slice architecture used in 7series Xilinx FPGAs, implementation of DSP algorithms using DSP slices enhance speed and efficiency of FPGA device within a small size and flexible hardware [11].

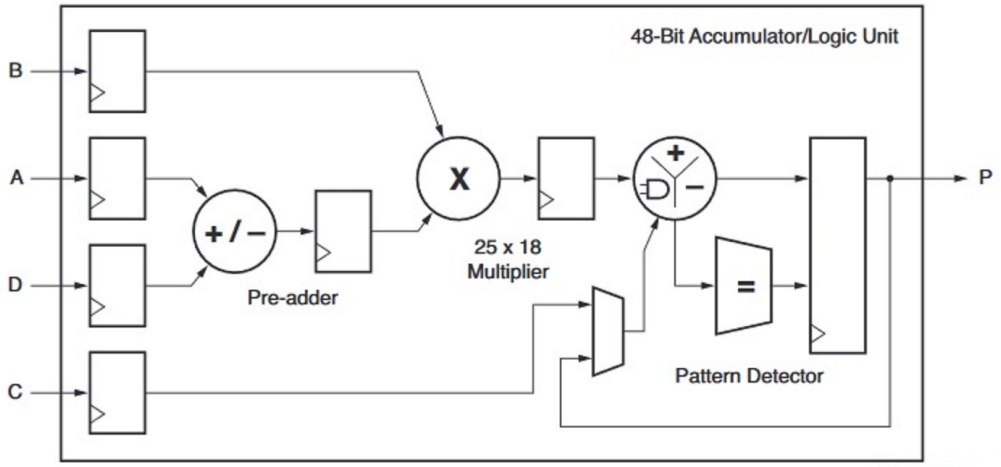


Figure 1.9 Slice of DSP

1.2.3 Memory Resources

Available on-chip memory is essential for high performance computing, however, LUTs in FPGAs can be used as distributed RAM across FPGA device, but dedicated blocks of RAM in FPGAs are the main resource of on-chip memory (fig 1.10). Dual 36 Kb block RAM can be configured as a 64K x 1 (when cascaded with an adjacent 36 Kb block RAM),32K x 1, 16K x 2, 8K x 4, 4K x 9, 2K x 18, 1K x 36, or 512 x 72 in simple dual-port mode [12].
In order to address performance and power requirement of different applications, Xilinx FPGAs are divided into three main families, table1.1 reports available resources for each family member, however, they have similar architecture, but performance and power metrics of each device is different mainly because of available resource and external memory band-width.

Table 1.1 7 Series FPGA Family comparison

MAXIMUM CAPABILITY	ARTIX	KINTEX	VIRTEX
LOGIC CELLS	215K	478K	1,955K
BLOCK RAM	13 Mb	34 Mb	68 Mb
DSP SLICES	740	1,920	3,600
MEMORY INTERFACE	1,066 Mbps	1,866 Mbs	1866 Mbps

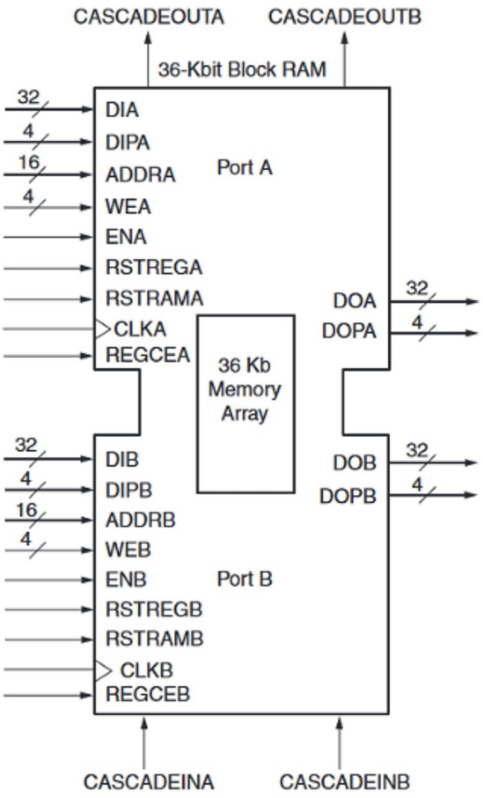


Figure 1.10 True Dual-Port Data Flows for a RAMB36

Figure 1.11 draws a comparison between three family members of Xilinx, each can be deployed based on application requirements. VIRTEX with maximum number of logic cells is designed for high performance applications, while ARTIX provides users with low power devices, moreover, KINTEX devices are manufactured to offer best performance-per-cost FPGAs to designers.

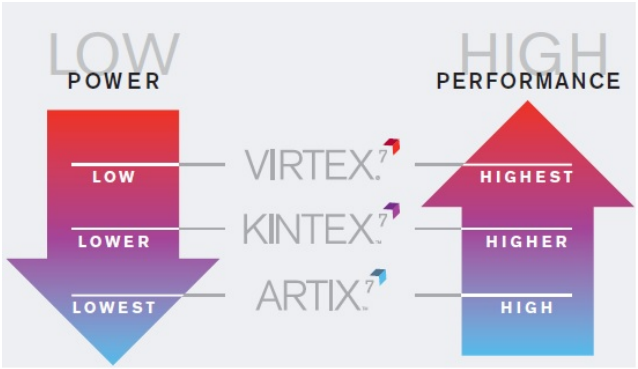


Figure 1.11 Comparison of 7 Series FPGA

INTEL FPGA Devices

However, Xilinx FPGAs are dominant in programmable logic market, but Altera FPGAs are also powerful devices with multiple classes that can offer costumers various performance, power and cost solutions to addressee different applications requirements. Additionally, reports in [13] suggest higher core performance of Intel FPGAs with respect to Xilinx UltraScale devices manufactured in 20 nm technology using ten different test bench targeting Intel and FPGA devices. Following lines briefly introduce five different FPGA classes produced by Intel which are currently closest competitor to Xilinx and may gain larger portion of FPGA market thanks to Intel investment and support[14].

STRATIX®

The Stratix® Series of FPGAs and SoC FPGAs are designed for the most demanding systems **where performance is paramount**. Stratix 10,figure 1.12 , is the flagship of Intel configurable device. Manufactured on Intel **14 nm Tri-Gate technology**, Stratix 10 will offer industry-leading capacity, performance, and architectural innovation for the most challenging computing, signal-processing, and software-defined networking applications.

Figure 1.12 STRATIX® 10 FPGAs developed by Intel

Arria®

The Arria® Series of FPGAs and SoC FPGAs deliver a **balance of performance and power efficiency**. Arria 10 FPGAs and SoC FPGAs are the latest product within the Arria family. Arria 10 at 20 nm has a unique combination of speed, DSP performance, capacity and power efficiency, and are the only **20 nm** FPGA to integrate an embedded processor system.

MAX®

The MAX® Series of programmable logic devices features a **non-volatile** architecture and offers a **low cost and low power configurable logics**. MAX devices are widely used for **general-purpose and power-sensitive** designs in a wide variety of market segments to perform functions that include **I/O expansion, interface bridging, power management, and FPGA configuration control**.

Cyclone®

The Cyclone® Series of FPGAs and SoC FPGAs are optimized for **low-cost, high-volume systems**. Cyclone V deliver capacity, performance, and IP ideal for the majority of embedded applications used in the **industrial and automotive markets**.

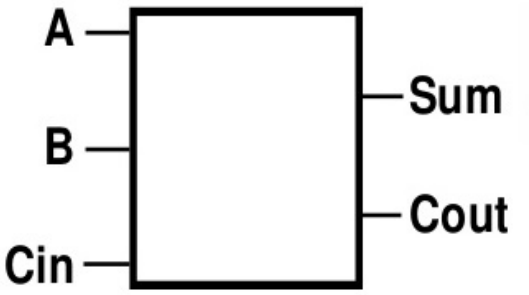
Enpirion®

The Enpirion® product line offers the industry's **most compact, energy-efficient, and sophisticated DC-DC converters** for meeting the power requirements of FPGAs. When power rails demand programmable **on/off, fast transient response and extra-low noise** devices they are the winner candidate.

1.3 HDL - Hardware Description Languages

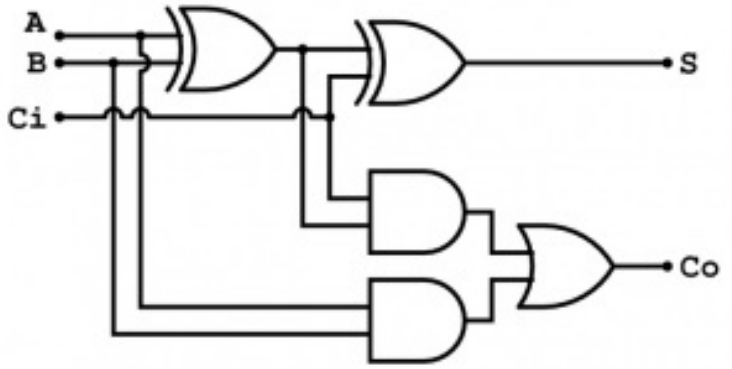
Hardware Description Language(HDL) is used to model digital electronic system, in other words, any digital electronic component such as registers, memories and switches can be modeled in HDL. VHDL/Verilog are common HDL languages in industry which support designing at three main levels of abstraction, although, designing digital electronic circuits can be done both using top-down and bottom-up approach, but the latter method is almost obsolete because of modern digital circuit complexity. Three different levels of HDL are shown in figure1.13 and described below.

- Behavioral Level: enable hardware designers to model whole algorithm starting from top level, each block consists of sequential function and tasks with defined inputs and outputs. In this level designers are focused on behavior and performance of the algorithm and only when behavioral level is verified, designers can develop underlying digital circuits of algorithm using register transfer level abstraction.
- Register-Transfer Level (RTL): In this level designers can model digital circuits at resister level using cycle accurate timing. Technically, RTL code can be used by synthesis tools in order to generate gate-level model from algorithm.
- Gate Level: The lowest level of abstraction is gate level which can be used to implement digital algorithm on silicon after placement and routing stage which can be done automatically by modern synthesis tools.



(a) Behavioral Scheme

(b) RTL Scheme



(c) Gate Level Presentation

Figure 1.13 Graphical Presentation of full adder in different HDL level

1.4 Contributions and Motivations:

The rest of this dissertation presents a study on the high-level synthesis (HLS) methodology for high performance computing applications (HPC) using two different approaches. To do so, firstly, this work discusses high-level synthesis, Model Based Design (MBD) and parallel computing that helps to provide the necessary background for analyzing the conducted experiments in this work. The main motivation behind this study relies on the importance of high performance and low power hardware to address the ever-growing demand for energy efficient platforms. Hence, the experiments are designed to examine Xilinx high-level synthesis tool chain in an analytical manner using various type of C based model (e.g. C, SystemC and OpenCL) for DSP and data base applications.

As the first stage of this PhD course, Simulink is used to develop a real-time application of digital signal processing unit of frequency modulated continuous wave (FMCW) radar. The top level model is used to generate functionality verified C and HDL based model in customizable fashion. Thereupon, the Xilinx synthesis tool chain is used to map automatic generated codes on FPGAs. Although, the results suggest that C based FPGA programming consumes less resources for performing the same amount of computation, performance hungry part of the DSP algorithm is substituted with SystemC IP that can be integrated within Simulink environment and be used to realize parallel RTL. The results of this study are discussed in more detail in chapters three, four and [15].

The growing trend towards heterogeneous platforms is crucial to meet time and power consumption constraints for high-performance computing applications. The OpenCL parallel programming language and framework enable programming CPU, GPU and recently FPGAs using the same source code. This eases software developers to implement applications on various devices supported by heterogeneous HPC platforms. This work presents two very different FPGA implementations of a database join operation, one using a direct O(n2) algorithm, and the other using a bitonic sort network to speed up the join operation. Comparison of performance and energy consumption for both FPGA and GPUs is provided which suggests an average of 40% performance-per-watt improvement by using an FPGA instead of a GPU. Extensive analysis and discussions related to these experiments are presented in chapters two, three and [16, 17]

Even though, MBD offers faster development cycle with respect to code based model due to early stage verification and coarse-grained subsystem integration, the available design space is much larger in latter case thanks to low level exposure capabilities. This leads the quest for the best implementation towards design space exploration (DSE) analysis of OpenCL models. The proposed DSE technique, discussed in the next chapter and demonstrated in the chapter four, offers careful design space exploration by considering on-chip and off-chip sources of parallelism. Additionally, power estimation based on the area utilization enables exploration of design space at a faster pace.

Chapter 2

HLS - High Level Synthesis

Technically speaking, High Level Synthesis(HLS) is automated flow which transforms C based model algorithm to register transfer level (RTL) that can be used to program FPGAs. Current Xilinx HLS tools chain supports C/C++, systemC and OpenCL as source code and generates high quality RTL based on design constraints and optimization directives provided by a designer [18, 19].

2.1 State Of The Art

Design complexity of electronic devices is growing day by day and manufacturing optimized hardware requires strong technology and knowledge to address the need of wide range of computational intensive workloads such as artificial intelligence(AI) and digital signal processing applications. Companies like Intel and NVIDIA manufacture parallel fixed-architecture GPUs which can be programmed by software developers using parallel programming languages such as OpenCL and CUDA. Moreover, the growing trend toward hybrid platform is crucial to meet time and power constraints of ever-evolving high performance computing applications.

Recent studies suggest that FPGAs have performance and power consumption advantage over GPUs thanks to technology advancement in FPGA manufacturing and development environment improvement which shortened the gap between hardware and software layer[20, 17]. Algorithms can be developed and verified at system level within the same frame work as CPU/GPU, this automated flow and FPGAs inherent reconfigurable parallel architecture result in high performance and low

power hardware which can be tuned and adopted according to design constraints during design cycles (figure 2.1).

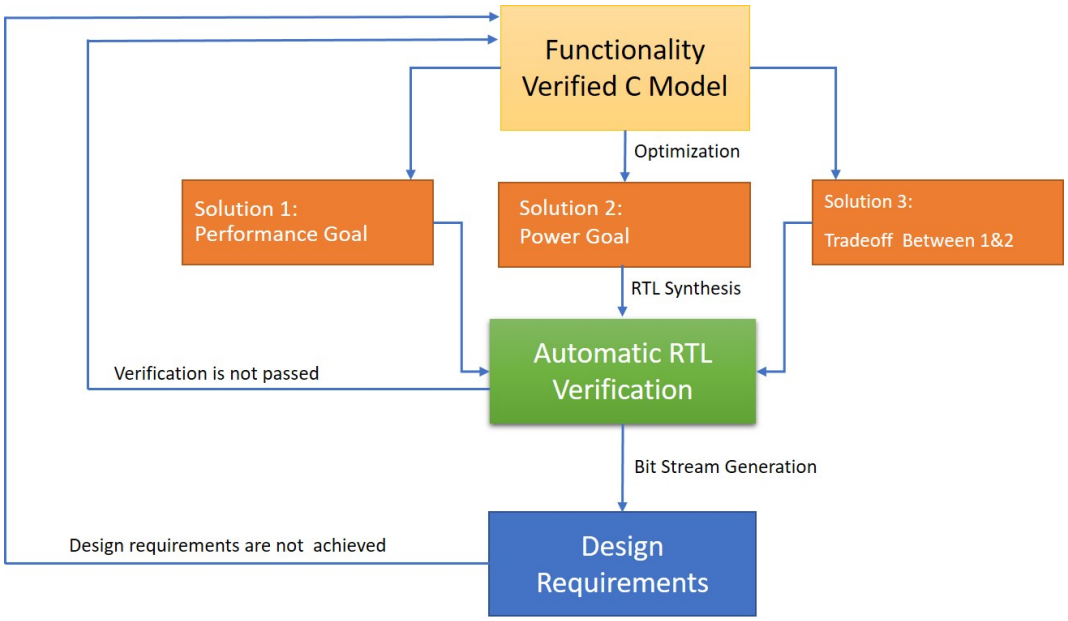


Figure 2.1 High level synthesis flow

High level synthesis enables designers with minimum hardware experience to program modern FPGAs through sophisticated tool chain that supports application development for FPGA starting from pure software level. Next section covers high level synthesis principles and optimization techniques which are used in this work to generate desirable RTL micro-architecture from higher abstraction layer.

High Level Synthesis Principles:
In this part of the thesis techniques and terminologies of Xilinx tool chain are explained using user manual and optimization instructions. The codes and figures are derived from Xilinx documentation which are prepared to introduce their product in a most advantageous form. However, each section refer to related document, it would be helpful to mention that the information of the rest of this section are obtained from [21–25].
Scheduling and binding are the processes at the heart of high level synthesis. The code below is used to explain theses process [24].

```
int foo(char x, char a, char b, char c) {
char y;
y = x*a+b+c;
return y
}
```

Scheduling: Scheduling is the process that HLS decides in which clock cycle operations should occur. This depends on the clock frequency, timing information of the target FPGA , and any additional optimization directives.

Binding: In this stage of high level synthesis, the tool determines which hardware resources implement each scheduled operation.

The scheduling phase section of figure 2.2 depicts this step. The multiplication and the first addition are performed in the first clock cycle. The next clock cycle executes the second addition. The green square in this figure indicates when an on-chip memory is written. The maximum number of scheduled operation in one clock cycle depends upon the clock period and the operation execution time. Faster FPGA can complete all above computations in one clock cycle, conversely, slower FPGAs may take more clock cycles.

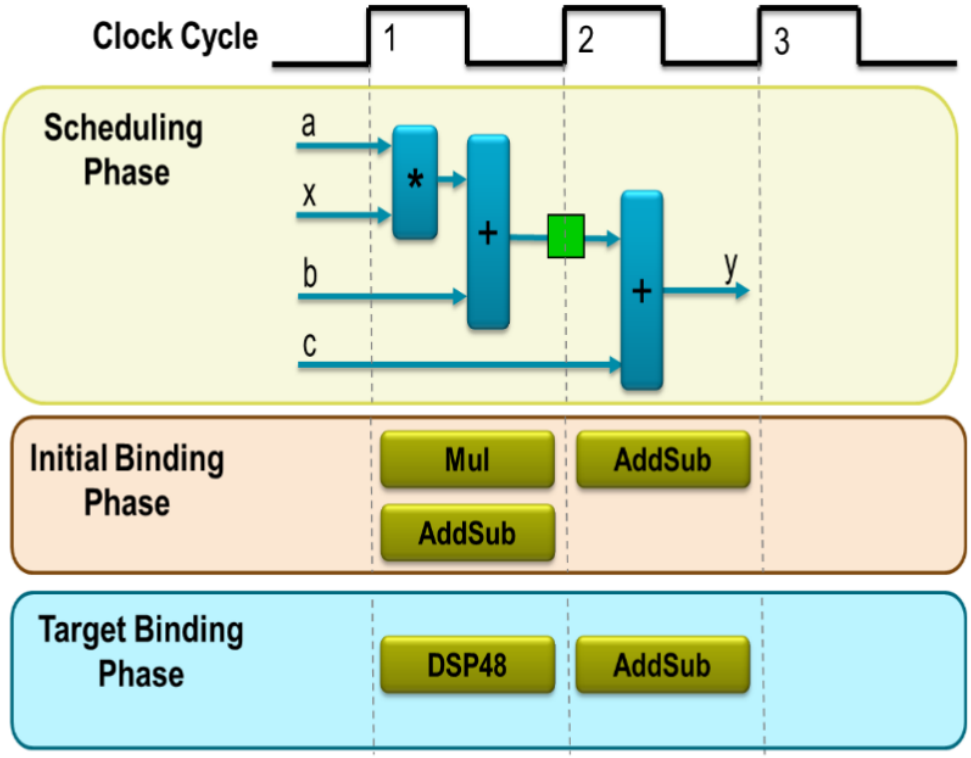


Figure 2.2 Scheduling and Binding

Specific information about FPGA target device is needed to decide which resource is the most optimized solution for each implementation. Multiplication and addition can be implemented by DSP48 resource which is the high performance and power-efficient unit in FPGA architecture [24].

Final step of high level synthesis is extraction and implementation of control logic and I/O ports. The simple C code below is used to clarify this process with more details. Generated RTL by HLS runs the logics inside the loop three times, high level synthesis tools produces a Finite State Machine (FSM) in the hardware design to complete these operations.

```
void foo(int in[3], char a, char b, char c, int out[3]) {
int x,y;
for(int i = 0; i < 3; i++) {
x = in[i];
y = a*x + b + c;
out[i] = y;
}
}
```

Figure 2.3 illustrates final scheduled design and generated FSM by HLS which is described below in more details.

- The C0 is the first state of FSM which is followed by C1, C2 and C3. The full sequence of states are :C0,{C1, C2, C3} ,{C1,C2,C3},{C1,C2,C3} , C0.
- The variables only once require the addition of the *b* and *c*. This operation pulls outside the (for-)loop and performed in the state *C0*. Each time the design enters state *C3* it reuses the operation result.
- The data for *in* is returned from the block-RAM in state *C2* and stored as variable *x*.
- The data from port *a* is read with other values to perform the calculation. The first *y* output is generated and the FSM ensures that the correct address and control signals are generated to store this value outside the block.
- The design then returns to state *C1* to read the next value from the array/block-RAM *in*.
- This process continues until all output is written.

- The design then returns to state *C0* to read the next values of *b* and *c* to start the process all over again.

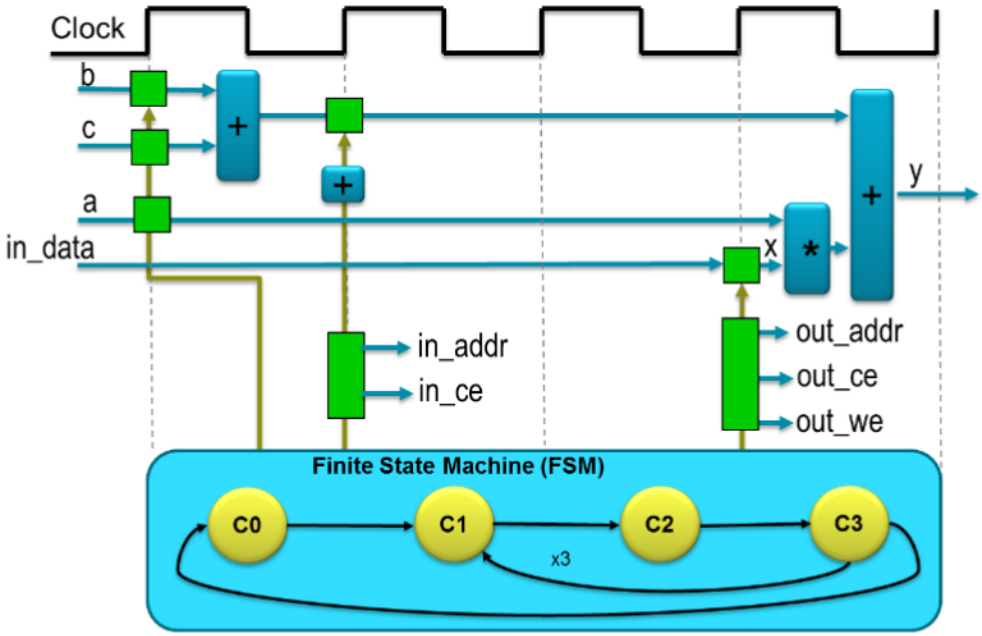


Figure 2.3 Control Extraction and IO port Sequencing

2.2 High Level Synthesis and Model Based Design

Model Based Design environments (MBDEs), such as Simulink, are becoming more widespread as they expand their capabilities of synthesizing efficient hardware and software from high-level algorithmic models. They find applications in very important areas such as digital signal processing (DSP), telecommunications, and control systems. MBDEs allow modeling of complex algorithms and systems at a very abstract level, using pre-defined primitive micro and macro blocks (e.g. adders, multipliers, multiplexers, FIR filters, FFTs). The designer can thus focus on defining the best algorithm without caring for tedious low level implementation details. Such details can be introduced later in the design flow via automated model-to-model translation, including both direct mapping and sophisticated hardware and software synthesis algorithms, or through a user-directed refinement process.

One of the major advantages of MBD tools is that they let the designer verify and validate abstract golden models against their design specifications. The designer can then use these models to generate code targeting either a specific embedded processor for software implementation, or a register transfer level (RTL) description for hardware synthesis. The process of generating software code or the RTL hardware description is normally assisted by the designer by providing constraints and directives. Algorithmic design provides a much better scope for power, area and performance optimizations as compared to what can be achieved at lower levels. MBD also greatly eases the verification task by allowing one to re-use already verified macro blocks and more importantly by letting the designer use the same verified golden reference model throughout the complete design, verification and implementation flow.

State of the art MBD tools used in the industry can generate very efficient and optimized software code for different target processors, by using information about the target processor architecture. But hardware implementation essentially entails the generation of a cycle accurate RTL model from very abstract block level models that have no notion of clock cycles. Hence the set of choices is much broader, and the normal direct translation strategy used for software implementation is likely to fail. Current MBD tools, such as Simulink from the Mathworks, can generate a very limited set of hardware implementations starting from a given model. In other words, they have limited capabilities to explore the hardware design space starting from a single model, due to reasons that are described more with detail in the next sections.

This is particularly true in the case of complex blocks like a Fast Fourier Transform (FFT), a Discrete Cosine Transform (DCT) or a Viterbi decoder, which are normally represented as Simulink macro blocks.

Simulink models can be also translated to RTL description for hardware synthesis through a tool called HDL Coder. Efficient hardware implementation starting from an abstract model generally requires effective design space exploration (DSE) from a single model. HDL Coder, however, has limited capabilities in this regard, especially when it comes to complex algorithms like FFT, DCT, and Viterbi decoders. Each HDL coder block is mapped to a few micro-architectures, e.g. fully sequential and fully pipelined, which provide only a few design points, such as minimum area or maximum throughput. Many of the architectural trade-offs that are essential for optimized hardware implementation, such as independent definition or throughput and latency, or the choice of memory parallelism and architecture, may even need to be performed manually, by changing the source model every time. This changing of model defies one of the main purposes of model-based design, by requiring different models for different implementations, and hence making the design process long and tedious.

Simulink has a rich library of components that can be used to model systems and algorithms from many different domains. In Simulink libraries, the components are arranged in groups known as blocksets, for example the DSP-blockset that can be used to model DSP algorithms. Simulink libraries are extensible through a mechanism known as S-functions. It provides a component modeling paradigm in which the functionality (algorithm) as well as the interaction with other components can be represented in a well-defined way. The S-functions can be written in C, FORTRAN, or MATLAB, as required.

Simulink comes integrated with a tool called Real Time Workshop (RTW). RTW is a set of code generators known as target language compilers (TLC) that can translate a Simulink model to C/C++. Each TLC can be optimized to generate code for a different processor or platform. Embedded Real Time (ERT) coder is one of these TLCs, which is optimized to generate software code for embedded applications. It can generate floating and fixed point code. **Part of this work presents a study on high level synthesis and the legacy code generated by RTW and HDL coder which can be divided into three main sections:**

1. Modeling and validation of digital signal processing unit of specific type of RADAR used in automotive industry via Simulink which will be described in chapter 3.

2. Generation of SystemC-based FFT IP with fully pipelined stages to be substituted with Simulink C model. The data flow and analysis of FFT algorithms are explained in chapter3.

3. Comparison of high level and RTL synthesis using HDL and C-based Simulink IPs is reported in chapter 4.

Alternative to model based design approach, in the next section, new programming framework is discussed that can target heterogeneous platforms equipped with CPU, GPU and FPGA devices. OpenCL(Open Computing Language) is supported by industry leaders to program various hardware accelerators within the same framework using close-to metal optimization options.

2.3 High Level Synthesis and OpenCL Model

The OpenCL Programming model has been developed by the Khronos group to overcome the hurdles of programming multi-core and heterogeneous compute platforms. OpenCL enables programmers to develop both close-to-the metal and portable software. Although, OpenCL is a high-level programming language, it provides a low-level abstraction layer that can expose significant architectural aspects of the target hardware, such as massive parallelism and the memory hierarchy. The CPU/GPU based platforms generally have a fixed architecture. While this makes programming easier and compilation times much faster, it is also a limitation because it reduces both the energy efficiency and the on-chip ("local" in OpenCL terms) memory access bandwidth with respect to an FPGA.

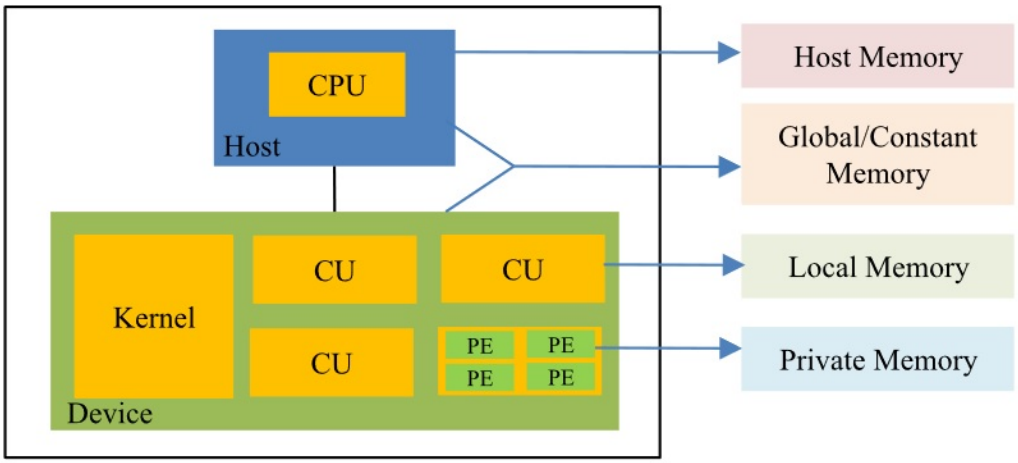


Figure 2.4 OpenCL platform and memory model

An OpenCL device consists of compute units (CU), each further divided into processing elements (PE) as shown in Fig 2.4. Several concurrent executions of the kernel body (called work-items) take place on multiple processing elements. The work-items are further grouped into work-groups, which are being executed by compute units. The memory is broadly divided into host (i.e. CPU) memory and device (i.e. GPU or FPGA) memory. The device memory is further divided into private memory (specific to each work-item), local mem- ory (shared by all the work-items in a work-group) and a global/constant memory (shared by all the work-groups). Access to global memory is the slowest (since it typically resides in external

DRAM) while private memory is the fastest (since it is typically allocated to register files), while local memory often resides in on-chip SRAM. Global memory however, is the largest in size while private memory is the smallest. The OpenCL memory model is also shown in figure 2.4. The work-items that compose an OpenCL kernel can be executed in an out-of order manner, in order to ensure high performance on a variety of platforms with different numbers of CUs [26]. Thus, the OpenCL standard uses a three-level synchronization and collaboration model. The execution order of different kernels is completely determined by the host code, either by calling them sequentially, or by using synchronization callbacks that notify the host code when a given kernel has completed execution. The execution of different work-groups within a kernel is completely unsynchronized, thus they must read and write different areas of global memory, and they cannot cooperate in any manner. Finally, the programmer can use explicit barriers to ensure local and global memory consistency for work-items within a work-group.A barrier represents a checkpoint within a work-group. All the work-items belonging to that work-group must reach it before any of them can proceed beyond it.

2.4 Xilinx SDAccel Development Environment

The new Xilinx SDAccel™ Development Environment provides high performance application developers the dedicated FPGA-based hardware and software tool chain. SDAccel offers a fast, micro and macro architecture optimized compiler that makes efficient use of on-chip FPGA resources; a well-known software-development flow with an Eclipse™-based Integrated Design Environment (IDE) for software development, profiling, and verification, which provides a CPU/GPU-like work environment; and dynamic reconfigurable accelerators optimized for different high performance applications that can be swapped in and out on the fly for a CPU/GPU-like run-time environment[22, 27].

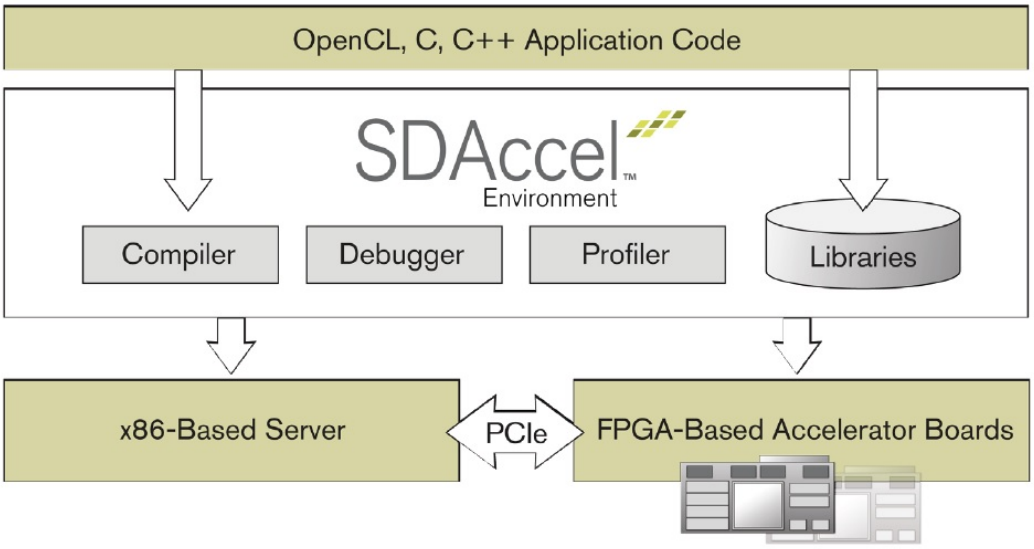


Figure 2.5 SDAccel CPU/GPU-Like development environment

OpenCL defines hierarchical memory model that is common between all vendors and can be applied to all OpenCL applications. Global, local and private memories are the main layers of this hierarchy. SDAccel maps them to the FPGA platform as external DRAMs, BRAMs, and register. SDAccel allows even finer-grained exploitation of the on-chip memory architecture of FPGAs by using directives such as on-chip global memory, multiple AXI buses for kernel global arrays, and partitioned local arrays, which enable a designer to finetune the memory architecture

and adapt the RTL architecture to the application, rather than the application to the GPU architecture[16].

The Xilinx® SDAccel OpenCL boards are PCIe® based accelerator cards that plug into a standard PCIe slot in x86_64 host or server type architectures.

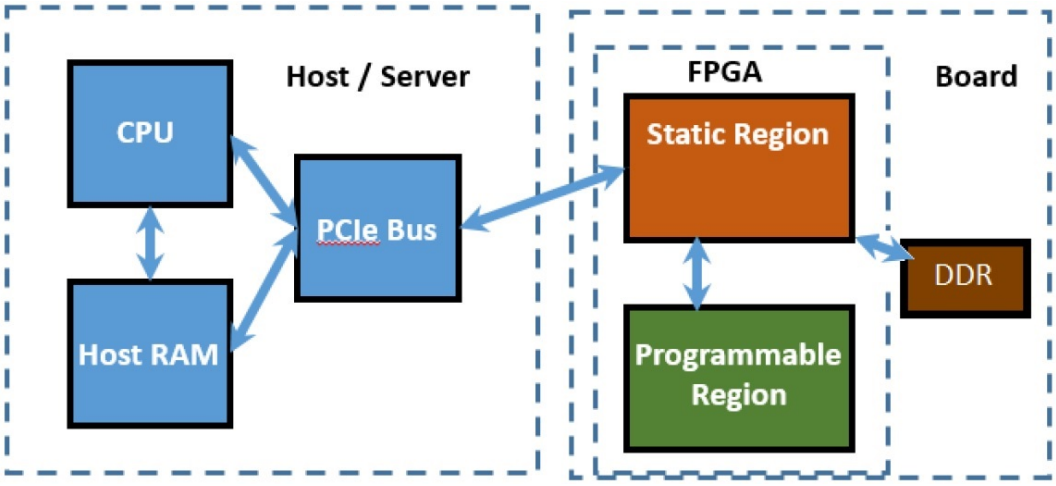


Figure 2.6 Programmable Device Block Diagram

The Xilinx PCIe hardware device consists of two regions, as shown in figure 2.6, the Static Region and the Programmable Region. The Static Region provides the connectivity framework to the Programmable Region, which will execute the hardware functions as defined in the software kernel. The diagram above is an example of a standard partial reconfiguration hardware platform. The expanded partial reconfiguration hardware platform would incorporate most of the static region into the programmable region [28].

The rest of this chapter is dedicated to fundamental concepts of host , static and programmable region of FPGA based platform and demonstrate how Xilinx FPGA device can be programmed in CPU/GPU-like environment using SDAccel tool chain.

2.4.1 Concepts of Application Host Code

In OpenCL framework, hostcode is responsible for manging the platform of accelerator mainly by performing following steps:

- Platform setup
- Allocate and transfer buffers to the device
- Run accelerator
- Read buffers from device memory

OpenCL programs are normally compiled completely at runtime which are handled by the host code APIs. Although, writing a proper host code for single or multiple kernels which can also act as test-bench will take considerable amount of time and programming effort, but application developers use template host codes and apply modifications based on algorithm requirements. Our group in Politecnico di Torino developed various design examples with different host codes for database applications for single and multiple kernels which are documented in public github repository and can be accessed by https://github.com/HLSpolito.

2.4.2 Static Region

In programmable device terminology, static region is containing all the necessary logics for implementation of **required interface between host, compute units and off-chip global memory**. This static region is a pre-defined base platform that can be flashed onto an EPROM on the board. The FPGA would then be programmed with this base platform upon power-up. As shown in Figure 2.7, communication between DDR, host and reconfigurable OpenCL region is performed through static region.

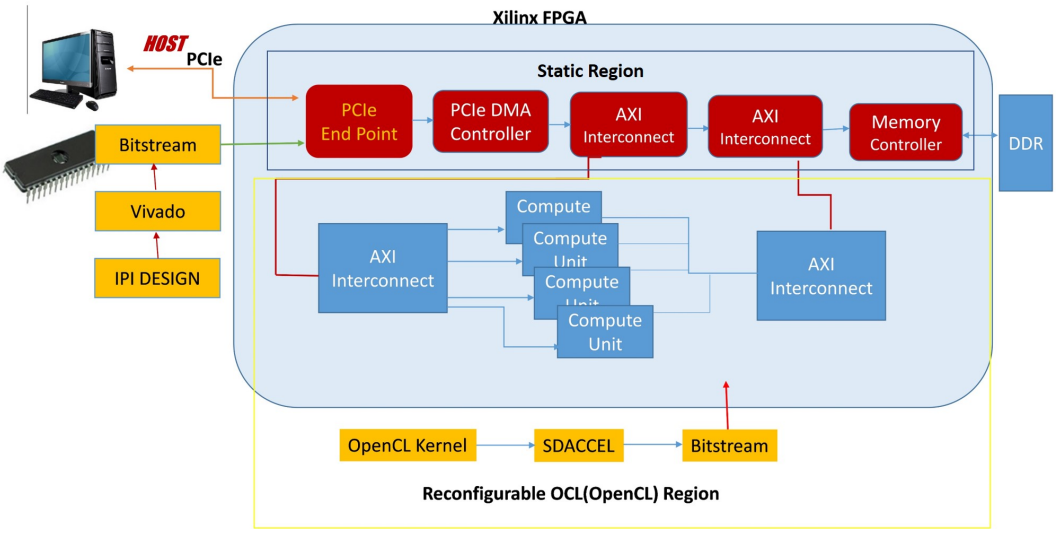


Figure 2.7 Block Diagram of Example Xilinx SDAccel Platform

2.4.3 Programmable Region

The programmable region contains the programmable section of the device that accepts the software kernel from the SDAccel tool chain. Implementation of computation intensive part of algorithm is done by optimizing and directing synthesis flow of OpenCL kernel using a wide range of provided directives by SDAccel. According to SDAccel terminology, area and performance optimization of the design falls into three main categories:

1. Host Code optimization
2. off-chip to on-chip interface optimization
3. On-chip optimization

In the following parts of this chapter, these categories are described in detail to discuss how one OpenCL source code can be implemented on FPGA with different power and performance characteristics using SDAccel development environment. To achieve the highest possible performance on FPGA these optimizations are paramount since the OpenCL standard guarantees functional portability but not performance portability.

Optimizing for an FPGA using the SDAccel tool chain requires the same effort as code optimization for a CPU/GPU. The one difference in optimization for these platforms is that in a CPU/GPU, the programmer is trying to get the best mapping of an application onto a fixed architecture. For an FPGA, the programmer is concerned with guiding the compiler to generate optimized compute architecture for each accelerator (referred to as a kernel) in the application.

As specified by the OpenCL standard, any code that complies with the OpenCL specification is functionally portable and will execute on any computing platform that supports the standard. Therefore, any code changes are for performance optimization. To aid the user in these optimizations, SDAccel offers performance profiling capabilities integrated into the run-time. This profiling helps the user analyze the achieved performance and pinpoint any potential bottlenecks that need to be addressed.

The SDAccel™ Environment is a complete software development environment for creating, compiling, and optimizing OpenCL™ applications to be accelerated on Xilinx® FPGAs. Figure 2.8 shows the recommended flow for optimizing an application in the SDAccel Environment [22].

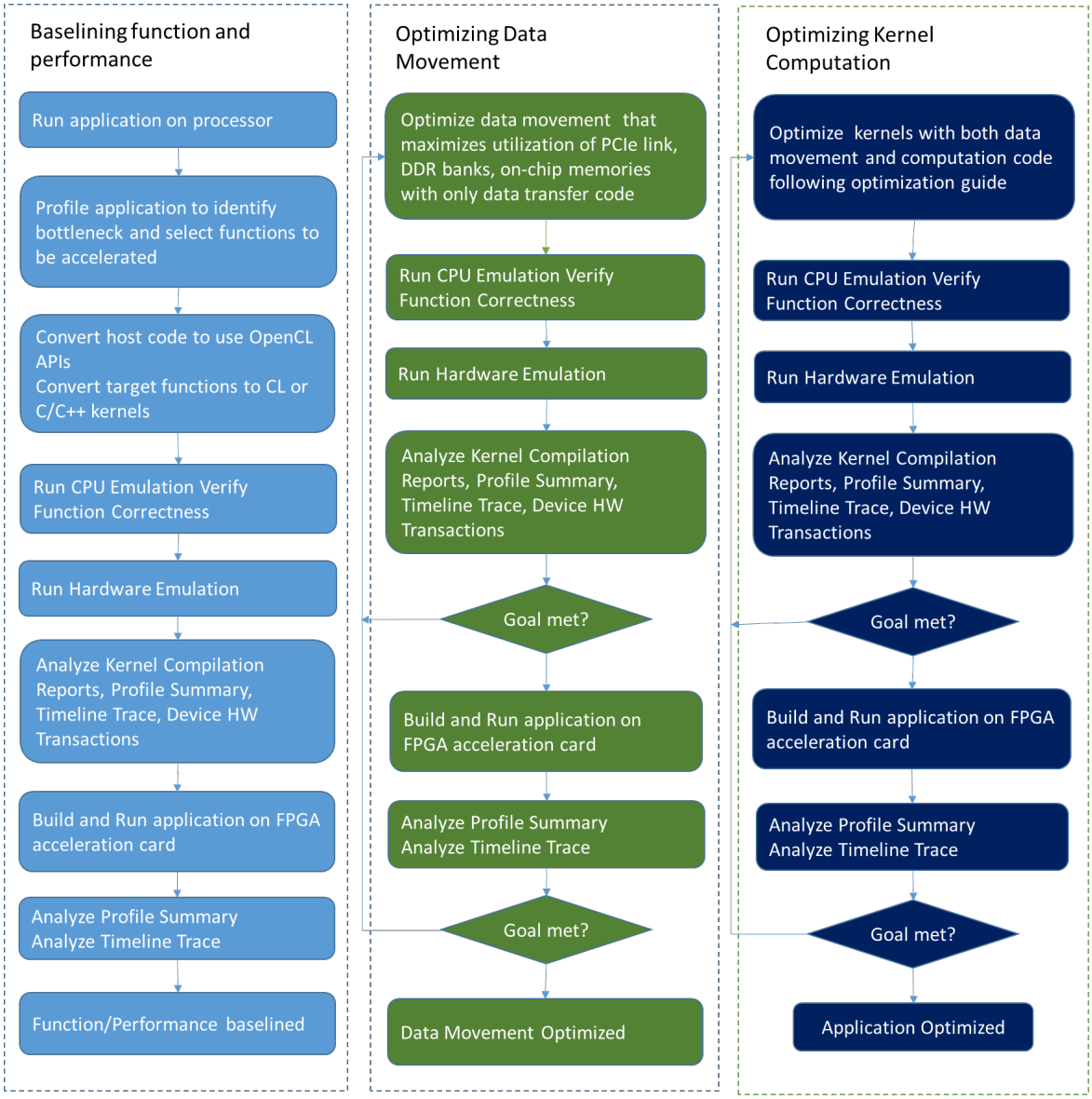


Figure 2.8 SDAccel Recommended Flow

2.4.4 Off-chip to On-chip Interface Optimization

In the OpenCL™ programming model, all data are transferred from the host memory to the global memory on the device first and then from the global memory to the kernel for computation. The computation results are written back from the kernel to the global memory and lastly from the global memory to the host memory. How data can be efficiently moved around in this programming model is a key factor for

determining strategies for kernel computation optimization, so it is recommended to optimize the data movement in your application before taking on optimizing the computation.

During data movement optimization, it is important to isolate data transfer code from computation code because inefficiency in computation may cause stalls in data movement. Xilinx recommends that you modify the host code and kernels with data transfer code only for this optimization step. The goal is to maximize the system level data throughput by maximizing PCIe bandwidth utilization and DDR bandwidth utilization. It usually takes multiple iterations of running CPU emulation, hardware emulation, as well as execution on FPGAs to achieve the goal.

Efficient data movement between the kernel running in the FPGA and external global memory is critical to the performance of acceleration applications. There is an inherent latency overhead to read and write data from external DDR SDRAM. A well-designed kernel minimizes this latency impact and maximizes the usage of the available data bandwidth provided by the acceleration platform.

SDAccel Environment includes a variety of FPGA acceleration cards with different DDR memory configurations. The figure below shows the data path between a kernel and one of 4 DDR channels on the XIL-ACCEL-RD-KU115 card. Each DDR channel provides 20GB/s raw DDR bandwidth with 80GB/s total for the entire card.

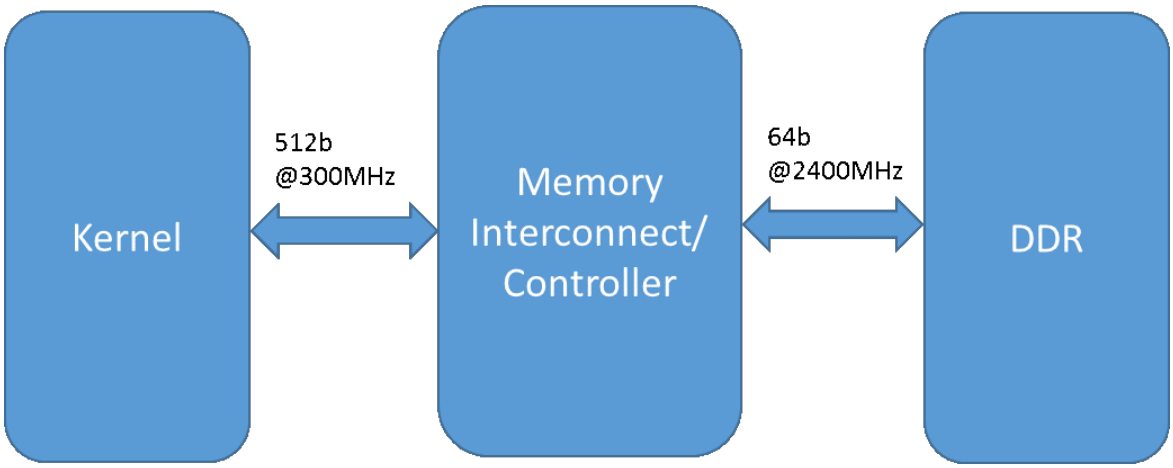


Figure 2.9 Data-width Configuration of Kintex7 Card

The width of the data path between the kernel and the memory interconnect/controller can configured by the SDAccel compiler as 32, 64, 128, 256, and 512 bits depending on the kernel argument types. For applications that require maximum data bandwidth between the kernel and DDR memory it is recommended that global pointers are defined explicitly as 512-bit data types.

OpenCL C specification defines vector data types that can have up to 16 elements of the same basic C data type. Kernel arguments defined as **int16**, **uint16**, and **float16** are automatically packed by the SDAccel compiler as 512-bit interfaces during synthesis.

Below is the code using **uint16** data type that directs synthesis toward wide memory interface between kernel and off-chip global memory. A 512-bit AXI4 memory mapped interface will be generated for these global pointers after compilation [25].

```
kernel __attribute__ ((reqd_work_group_size(1, 1, 1)))
void vadd(
const __global uint16 *in1, // Read-Only Vector 1
const __global uint16 *in2, // Read-Only Vector 2
__global uint16 *out, // Output Result
int size // Size in integer
)

{
KERNEL BODY
}
```

Inferring Burst Transfer from/to Global Memory:

The most common global memories used on Xilinx® acceleration cards are DDR3 and DDR4 SDRAMs. They are most efficient when operated in burst mode. In addition there are overheads associated with switching between DDR read and write. Xilinx recommends to transfer large amount of data in a single burst to achieve the best efficiency of the memory controller and keep the compute unit inside the FPGA device busy all the time.

The memory layout of data objects is a key factor to consider for improving the data transfer efficiency. Considering a 4x4 matrix "a" example, conceptually it is a two dimensional array as shown in the matrix logical layout in the Figure below. In C/C++ programming, arrays are physically stored in row-major order that all data within a row are stored in consecutive locations followed by the data within the next row as shown in the matrix physical layout below. The implication is that if your algorithm reads the data column-wise, the burst transfer will not happen as it reads from discrete location each time. This can generally be optimized by either transposing your data in the host code or caching multiple columns of data in the kernel [25].

Matrix Logical Layout

row/col	0	1	2	3
0	a[0][0]	a[0][1]	a[0][2]	a[0][3]
1	a[1][0]	a[1][1]	a[1][2]	a[1][3]
2	a[2][0]	a[2][1]	a[2][2]	a[2][3]
3	a[3][0]	a[3][1]	a[3][2]	a[3][3]

Matrix Physical Layout

addr	value
0	a[0][0]
1	a[0][1]
2	a[0][2]
3	a[0][3]
4	a[1][0]
5	a[1][1]
6	a[1][2]
7	a[1][3]
8	a[2][0]
9	a[2][1]
10	a[2][2]
11	a[2][3]
12	a[3][0]
13	a[3][1]
14	a[3][2]
15	a[3][3]

Figure 2.10 Memory Layout Matrix

Figure 2.10 depicts physical layout of memory access pattern on FPGA device that can be read from/to in burst fashion. Below is the code that suggests proper coding style for performing burst read for one dimensional vectors.

```
kernel __attribute__ ((reqd_work_group_size(1, 1, 1)))
void vadd(
const __global uint16 *in1, // Read-Only Vector 1
const __global uint16 *in2, // Read-Only Vector 2
__global uint16 *out, // Output Result
int size // Size in integer
)
{
local uint16 v1_local[LOCAL_MEM_SIZE]; // Local memory to store vector1
int size_in16 = (size-1) / VECTOR_SIZE + 1;
...
for(int i = 0; i < size_in16; i += LOCAL_MEM_SIZE)
{
...
int chunk_size = LOCAL_MEM_SIZE;
//boundary checks
if ((i + LOCAL_MEM_SIZE) > size_in16)
chunk_size = size_in16 - i;

v1_rd: __attribute__((xcl_pipeline_loop))
for (int j = 0 ; j < chunk_size; j++){
v1_local[j] = in1 [i + j];
}
...
}
}
```

The Device Hardware Transaction view below shows that multiple read bursts are sent at the kernel start and all read data come back continuously after the memory read latency.

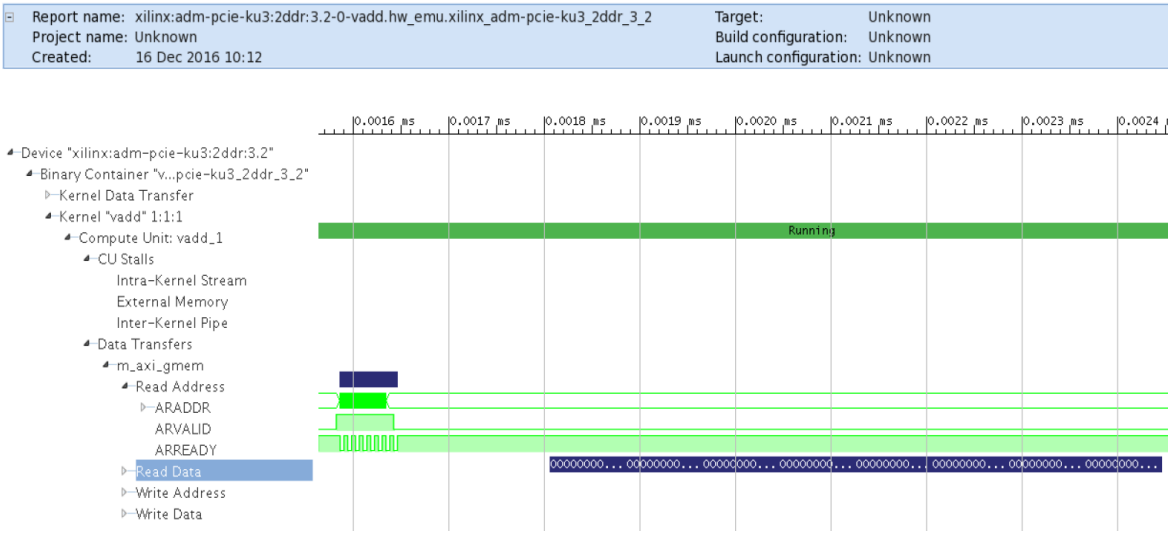


Figure 2.11 Device-Hardware-Transaction Timing Diagram

Using Multiple DDR Banks:

For applications demanding high bandwidth to the global memory, devices with multiple DDR banks can be targeted so that kernels can access all available memory banks simultaneously. For example, SDAccel™ includes platforms that support multiple DDR banks which is supported by Xilinx® vendor extension. Creation of multiple AXI4 interfaces are necessary to assign multiple DDR bank to each kernel which can be done both from host code or inside OpenCL kernel. Figure 2.12 depicts high level representation of two BANK implementation of OpenCL source code.

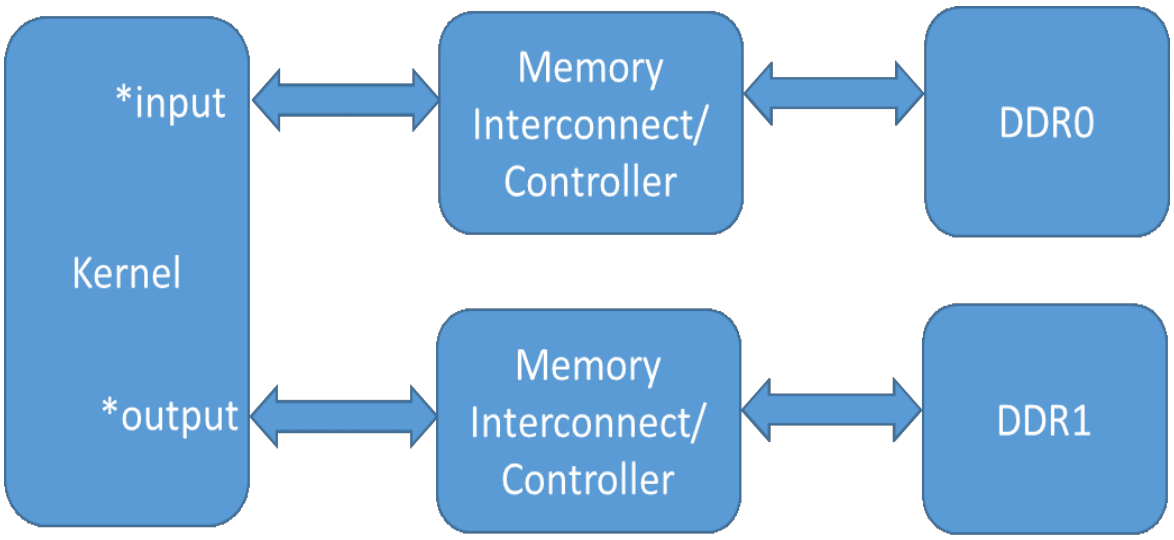


Figure 2.12 Off-chip memory with two separated Banks

Figure 2.13 shows transaction of off-chip memory with on-chip AXI interface using two separate DDR banks. Read and write operations are performed separately and in parallel which result in improvement of overall bandwidth utilization [25].

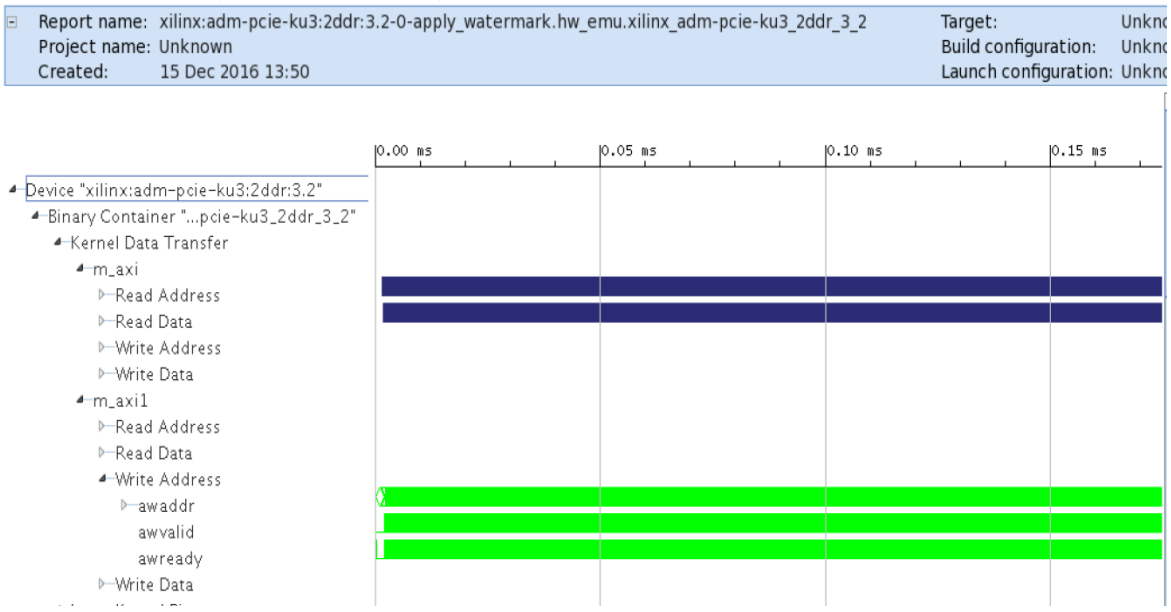


Figure 2.13 Device Hardware Transaction

2.4.5 On-chip optimization

Interface optimization is crucial and the first step of RTL optimization. After high speed interface synthesis that transfer data between on-chip and off-chip section of FPGA platform with maximum possible efficiency, data is processed on-chip in pipelined and parallel fashion. This requires to tailor and direct compilation flow of OpenCL and C/C++ kernel toward parallel and pipelined hardware using wide range of provided directives by Xilinx HLS tool. Table 2.1 and Table 2.2 list array and loop-level optimization directives in Xilinx high level synthesis tool which this work provides overall review over them. For additional information on other optimization techniques and how they can be implemented using Xilinx high level synthesis tools it is suggested to refer to provided user-guide by Xilinx [22, 24].

Table 2.1 Loop Level Optimizations

Unrolling	Unroll for-loops to create multiple independent operations
Merging	Merge consecutive loops to reduce overall latency,
Flattening	Allows nested loops to be collapsed into a single loop
Dataflow	Allows sequential loops to operate concurrently
Pipelining	Used to increase throughput by performing concurrent operations
Tripcount	Provides user override of iteration analysis
Latency	Specify a cycle latency for the loop operation

Table 2.2 Array Optimization

Resource	Specify which hardware resource (RAM component) an array maps to
Map	Reconfigures array dimensions
Partition	Partitioned into multiple smaller arrays
Reshape	Reshape an array from one with many elements to one with greater word-width
Stream	Specifies that an array should be implemented as a FIFO rather than RAM.

Unrolling Loops

By default loops are kept rolled in High-Level Synthesis. That is to say that the loops are treated as a single entity: all operations in the loop are implemented using the same hardware resources for iteration of the loop. Below is the code with simple for loop that perform multiplication between array b and c and store results in array a. Figure 2.14 shows presentation of three different hardware solutions for the same source code [24].

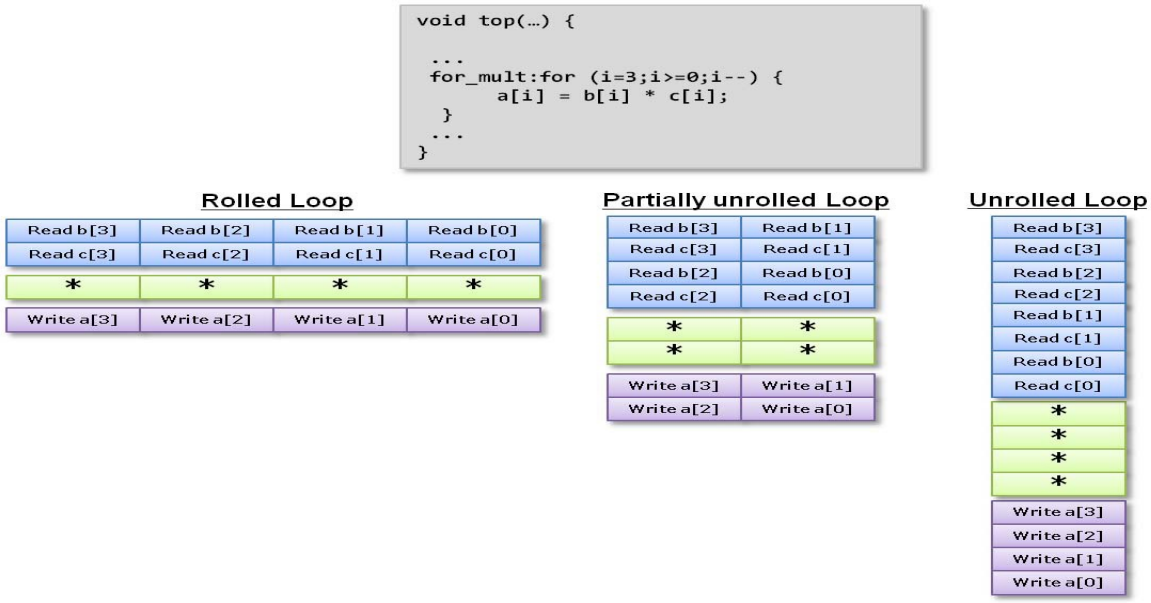


Figure 2.14 Loop Unrolling

- **Rolled Loop**: When the loop is rolled, each iteration will be performed in a separate clock cycle. This implementation takes four clock cycles, only requires one multiplier and each RAM can be a single port RAM.
- **Partially Unrolled Loop**: In this example, the loop is partially unrolled by a factor of 2. This implementation required two multipliers and dual-port RAMs to support two reads or writes to each RAM in the same clock cycle. This implementation does however only take 2 clock cycles to complete: twice the throughput and half the latency of the rolled loop version

- **Unrolled loop**: In the fully unrolled version the entire loop operation can be performed in a single clock cycle. This implementation however requires four multipliers. More importantly, this implementation requires the ability to perform 4 read and 4 write operations in the same clock cycle. Since quad-port RAMs are not common, this implementation may require the arrays be implemented as registers rather than RAMs.

Array partitioning

The message below is the common problem when pipeline directive is used. This issue avoid proper pipelining of generated hardware with **initiation interval(II) of one** which means successive iteration of a loop can not be executed next after each other with minimum latency [24].

> @*I* [SCHED-61] Pipelining loop 'SUM_LOOP'.
> @*W* [SCHED-69] Unable to schedule 'load' operation ('mem_load_2', bottleneck.c:57) on array 'mem' due to limited memory ports.
> @*I* [SCHED-61] Pipelining result: Target II: 1, Final II: 2, Depth: 3.

Arrays access conflicts usually cause this kind of problem. Implementation of arrays as block-RAM with maximum of two data ports can limit the throughput of a read/write operations within the loop.

Arrays are partitioned using the ARRAY_PARTITION directives in order to increase available memory port and overall bandwidth utilization.Vivado HLS enables designers to partition on-chip memory in three different styles as shown in figure 2.15 and described below .

- Block:The original array is split into equally sized blocks of consecutive elements of the original array.
- Cyclic:The original array is split into equally sized blocks interleaving the elements of the original array.

- Complete: The default operation is to split the array into its individual elements. This corresponds to resolving a memory into registers.

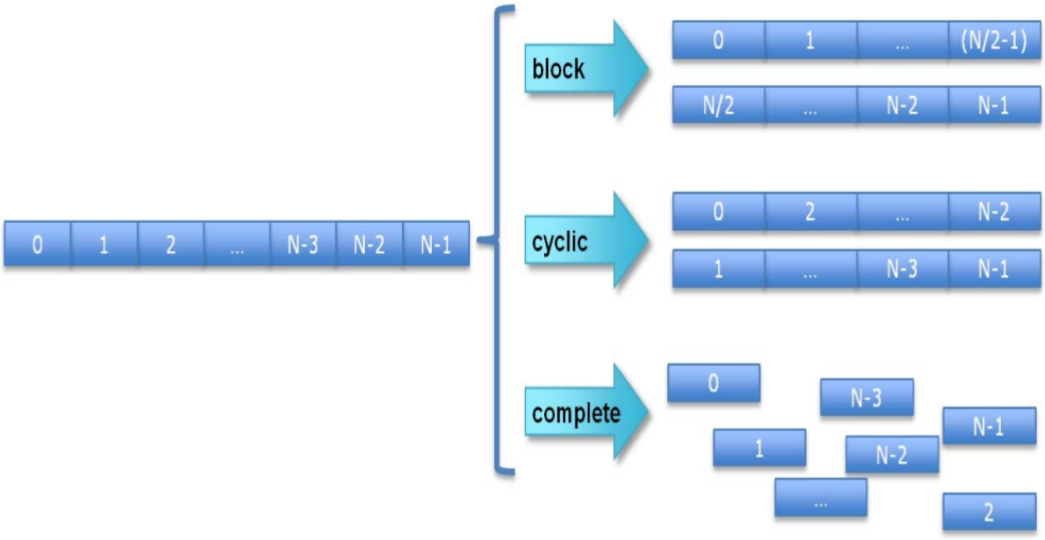


Figure 2.15 Array Partitioning

Array Mapping

Small arrays in source code can be mapped into a single larger array by map directive which usually reduces the number of required block-RAM [24].

The ARRAY_MAP directive supports two ways of mapping small arrays into a larger one:

- **Horizontal mapping**: This corresponds to creating a new array by concatenating the original arrays. Physically, this gets implemented as a **single array with more elements**. This option is useful to increase **resource sharing and minimizing power consumption**.
- **Vertical mapping**: This corresponds to creating a new array by concatenating the original words in the array. Physically, this gets implemented by a single

array with a **larger bit-width**. This option is useful to **increase memory bandwidth**.

2.5 Design Space Exploration and HLS

It has been a long endeavor and evolution in Electronic-System-Level design industry to generate and optimized automatic RTL from C-based description, nowadays, generation of high quality RTL for industrial purpose is possible thanks to modern HLS tool [29–31], more importantly, available directives such as loop unrolling, memory partitioning and multiple instantiation of compute kernels[1](multi-core) enable designers to perform broad and in-depth design space exploration(DSE) to realize application driven hardware. In fact, fine and coarse-grained parallelism of generated RTL are feasible by directing the compilation flow of C-based model using customizable optimization options provided by modern HLS tools to drive optimum solution with highest performance-per-watt.

The rest of this work covers design space exploration of related work to explain the orientations and approaches of previous HLS based DSE. Afterwards, the author proposes an automated DSE flow for multi core OpenCL based FPGA implementation which provides design hints based on Xilinx HLS output.

Various DSE approaches are studied and proposed in literature. A Pre-RTL power-performance simulator, called Aladdin, is proposed in [32] which enables rapid design space exploration of accelerator with high accuracy in the early stage of design. Author in [33] suggests that careful exploration of all solutions can result in area efficiency by proper partitioning of multidimensional arrays and unrolling nested-loops to reduce the area over head caused by bank switching. Automated DSE flow is proposed in [31] to obtain Pareto-optimal curve (performance versus area) of the application mapped on FPGA using HLS methodology, similarly, HLS-based DSE approach is discussed in [34] based on user defined area and time constraint which suggests the best RTL solution based on the design requirements. HLS enables designer to select the bit-width of variables from behavioral description, author in [35] describes a method to perform DSE for FPGA by controlling the amount of resource sharing using automatic bit-width controller of HLS tool. Parallel and multi-threaded method for finding the optimum micro-architecture for a given SystemC model is presented in [36], the author suggests a DSE flow to minimize the size of design for a given target latency.

[1] In this work, compute units in FPGA design resemble processing cores in CPU/GPU architecture

As HLS tools are becoming more mature, the demand of C-based IPs are increasing. Synthesizable C-based IPs require DSE at micro-architectural level, author in [37, 38] presents automated flow to perform design space exploration on generated C models from Simulink IP block sets which shields a designer to understand legacy code and obtains a set of Pareto-optimal solutions based on defined constraints, additionally, macro-architectural trade-offs are considered by wrapping different parts of the C-model into SystemC version to derive parallel RTL from behavioral description of Simulink model via HLS. Study in [39] presents a learning-based method for DSE that eases and accelerates micro-architectural modification using Random-Forest model which considers all different knobs (micro-architectural choice) and number of possible choices(e.g. unrolling and partitioning factor) to select Pareto-optimal solution for final RTL realization.

However, important aspects of on-chip DSE are discussed at above-mentioned works, but efficient data transfer from off-chip to on-chip memory is a design necessity to drive high performance RTL. Interestingly, architectural template is proposed in [40] that is capable to consider off-chip source of parallelism in deep convolutional neural networks in which author claims generated RTL has better performance in comparison to previous works using identical neural networks targeting same FPGA devices.

In the next section, we propose a design space exploration method considering area, power and execution time of each OpenCL kernel among set of candidate solutions. Our experiment suggests, reported in chapter four, our best RTL on Xilinx Virtex7 has higher performance-per-watt with respect to two different high-end GPUs manufactured in the same node (28 nm). Moreover, our golden solution can outperform tested GPUs by using Xilinx UltraScale FPGAs (20 nm) at the cost of more expensive devices with much larger available on-chip resources.

2.5.1 DSE of Multi-Core RTL via OpenCL Synthesis

In order to sift through design alternatives prior to implementation, analyzing performance and efficiency of off-chip to on-chip memory transfers is crucial. To do so, we introduce the following parameters and monitor memory interface performance using different configurations. Table 2.3 reports the memory interface performance

of the generated RTL by SDAccel for burst and normal memory transfers.

Number of Transfers (NT): Number of total independent transfers between DDR and kernel. Each transfer sends a pack of data with the specified burst size for read or write operations.

Burst Size (BS): The data size of each transfer. The maximum burst size is assumed to be 4096 bytes [23].

Transfer Efficiency (TE %): Burst size / Maximum Burst size.

Transfer Rate (MB/s): (Total Transfer (MB) / Total Transfer time (s))

Bandwidth Utilization (BW%): (Transfer Rate / Available Bandwidth [2])

Table 2.3 SDAccel off-chip to on-chip transfer analysis

	NT	BS	TE	BW	MB	Time(ms)
Burst	32	1024	25%	6.8%	.032768	.049
Non-Burst	8192	64	1.56%	61%	.524288	.088

Figure. 2.16 plots bandwidth utilization versus on-chip memory size. The non-burst memory configuration has low transfer efficiency, which increases the overall execution time of the kernel. In the second scenario, the memory controller is using burst access which increases transfer efficiency. Note that the total transfered data is much higher in the second row which explains why the transfer efficiency is different with respect to the burst mode. The results are important since they demonstrate the effect of local size on memory controller performance using available IPs for on-chip to off-chip interface.

[2]In this work we consider a 20 GB/s maximum available bandwidth for evaluation purposes.

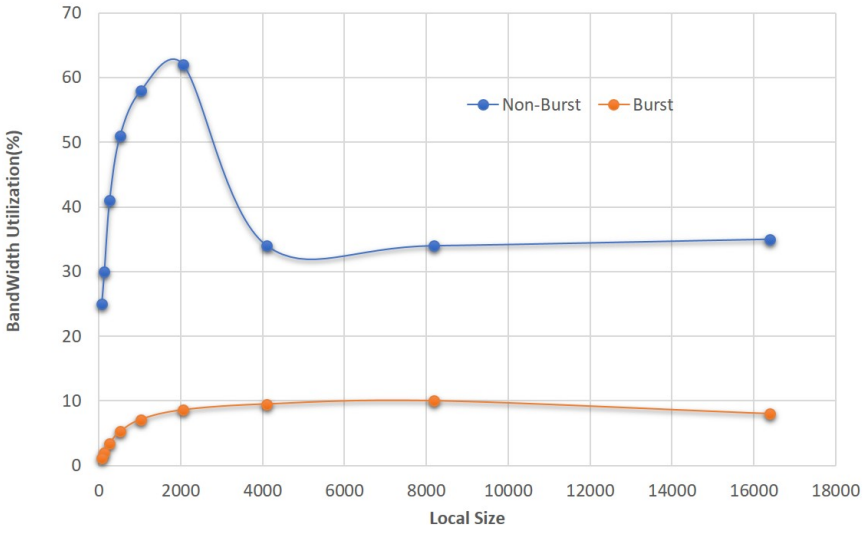


Figure 2.16 Off-Chip bandwidth utilization vs on-chip memory size

The results of this section are obtained only by performing read and write operations to evaluate the performance of memory controller by isolating on-chip computation from off-chip transfers. The remaining memory bandwidth could be used, for example, to increase the computational power of the FPGA, by instantiating multiple parallel compute units (OpenCL work-groups).

Algorithm 1 describes the design space exploration method used in this work. The algorithm below, illustrated in Figure. 2.18, suggests an automated flow which considers area utilization, execution time and total number of work-groups for each solution and suggests the minimum required numbers of compute-unit to meet the given time constraint. Figure. 2.20 shows the output of DSE flow for each kernel used in the sorting network. The bitonic-sorting algorithm is composed of three different kernels each with different off-chip memory access pattern and internal computations. **In fact, each solution is a RTL core with a unique optimization setting synthesized from the OpenCL WGs**. The solution with lowest energy consumption that meets resource and performance constraint is chosen for implementation of each kernel on the target FPGA.

The proposed algorithm estimates the quality of each solution considering three main criteria described below:

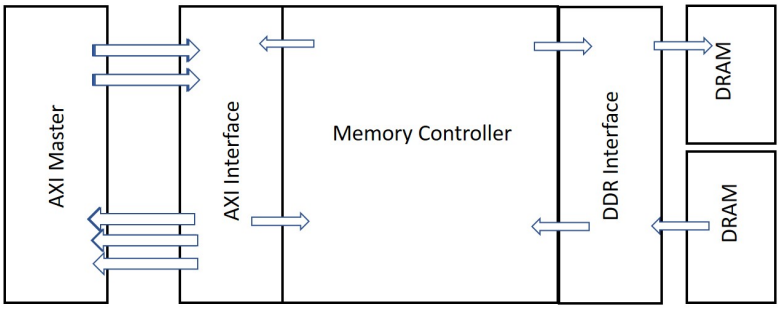


Figure 2.17 AXI memory controller diagram

1. Execution time of each compute unite obtained from cycle accurate RTL simulation. The performance of each RTL core , generated from OpenCL WG, depends on off-chip bandwidth utilization and on-chip data path optimization.

$$time[i][j]; \quad i \in \{1 \ldots Solution\}, \quad j \in \{1 \ldots Kernels\}$$

2. Design area of each solution that also reflects dynamic power consumption of the kernel. Figure. 2.19 illustrates the semi-linear relation between LUT utilization on a Virtex7 device and dynamic power consumption for the three kernels used in the bitonic-sort algorithm.

$$LUT[i][j]; \quad i \in \{1 \ldots Solution\}, \quad j \in \{1 \ldots Kernels\}$$

3. Total number of required compute units to complete computation which depends on the local size of the OpenCL kernel and may decrease the performance of the device due to inefficient usage of the memory controller[3].

The main motivation behind developing this routine is to estimate the relative quality of all solutions with respect to each other. Note that the quality of each solution reflects the execution time and power consumption of each RTL core. Considering a more general equation for estimating power consumption based on the area utilization and throughput of each OpenCL WG is left to future work.

[3]Memory controllers (Figure. 2.17) contain the logic necessary to read and write to DRAM, and to "refresh" the DRAM.

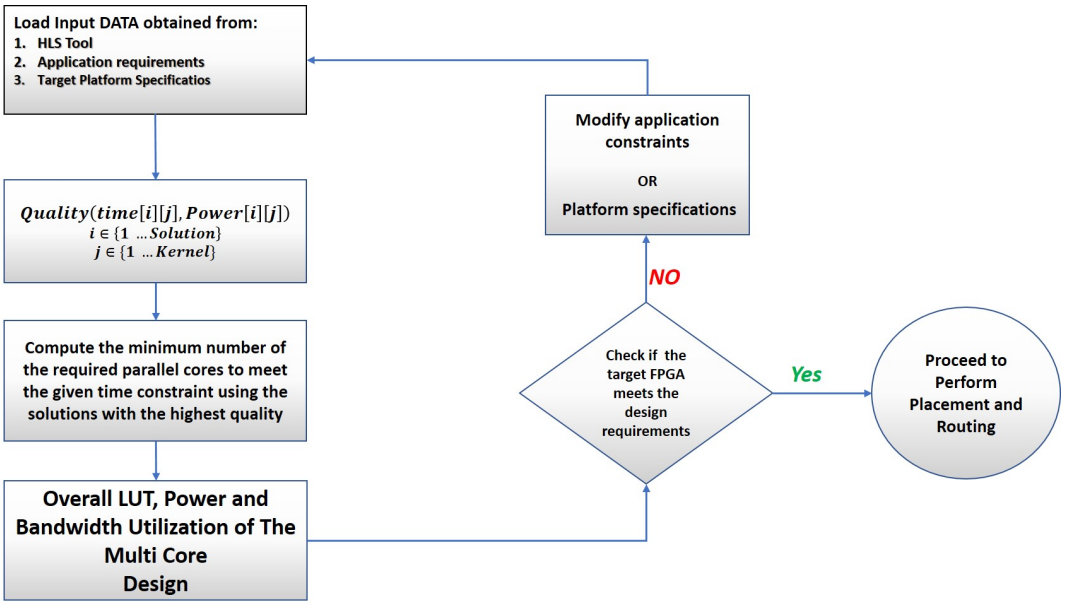


Figure 2.18 Design Space Exploration Flow of Multi Kernel OpenCL Models

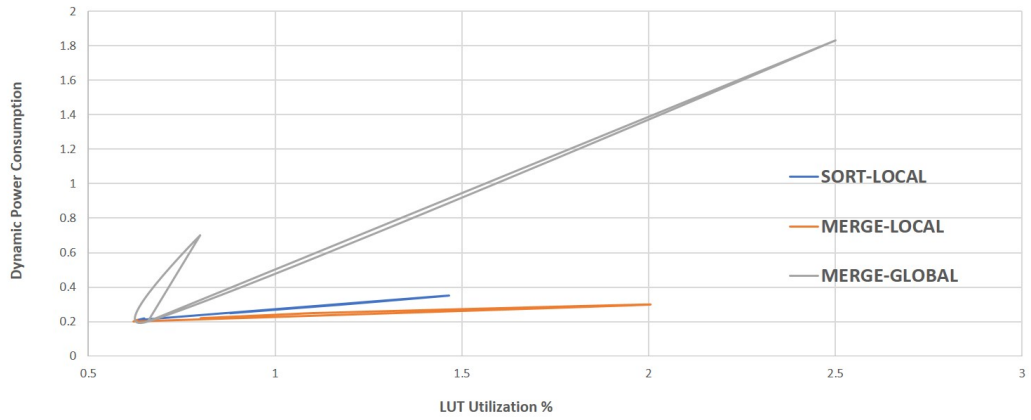


Figure 2.19 Dynamic power consumption versus LUT utilization

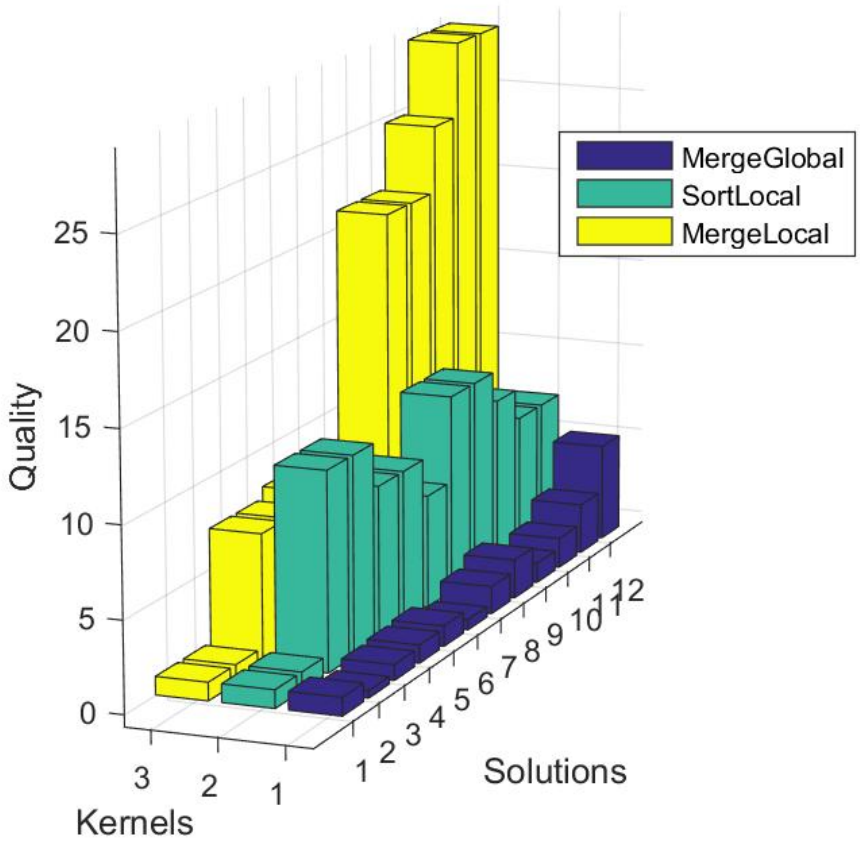


Figure 2.20 Quality of generated RTL by SDAccel for three kernels used in the sorting network

Algorithm 1: Design Space Exploration of Multi Kernel OpenCL Models

Input:
Solutions: LUT, Execution Time, Number of calls, Bandwidth utilization and Dynamic Power Consumption.

for *each* $i \in \{1 \ldots number\_of\_solutions\}$ **do**
  for *each* $j \in \{1 \ldots number\_of\_kernels\}$ **do**
    load $LUT[i][j]$; *LUT utilization of solution=i and kernel=j*
    load $time[i][j]$; *Execution time of solution=i and kernel=j*
    load $call\_number$[i][j]; *Total number of calls for solution=i and kernel=j*
    load $BW[i][j]$; *Off-chip bandwidth utilization of solution=i and kernel=j*
    load $P_d[i][j]$; *Dynamic power consumption of solution=i and kernel=j*
  end
end

Application: Time constraint
Platform: Available LUT , Available off-chip bandwidth

Output:
Optimum solution for each kernel, Minimum number of parallel compute-units (cores) to meet the time constraint,

Total LUT, Bandwidth and Dynamic power utilization of the best solution for multi core RTL

Begin

for *each* $i \in \{1 \ldots number\_of\_solutions\}$ **do**
  for *each* $j \in \{1 \ldots number\_of\_kernels\}$ **do**

$$Quality[i][j] = (time[1][j]/time[i][j]) * (LUT[1][j]/LUT[i][j]) * (call\_number[1][j]/call\_number[i][j]);$$

  end
end

for *each* $j \in \{1 \ldots number\_of\_kernels\}$ **do**

$$M[j] = Maximum(Quality[:][j]);$$

Choose the solution with the highest quality for each kernel, Maximum function stores the maximum value index
end

$$Total\_time = \sum_{j=1}^{kernels} call\_number[M[j]][j] * time[M[j]][j];$$

$$NC = Total\_time/Time\_constraint;$$

Minimum number of required compute-unit is calculated based on the quality assessments

$$Total\_P_d = \sum_{j=1}^{kernels} NC * P_d[M[j]][j];$$

$$Total\_LUT = \sum_{j=1}^{kernels} NC * LUT[M[j]][j];$$

$$Total\_BW = \sum_{j=1}^{kernels} NC * BW[M[j]][j];$$

if $Total\_LUT < 60\%\ Available\_LUT$ *and* $Total\_BW < 80\%\ Available\_BW$ **then**
    The design fits neatly into the Target FPGA.
end if
else
    The Platform does not satisfy the design requirements.
end
End

Chapter 3

HPC - High Performance Computing

High Performance Computing(HPC) was traditionally used by governments and universities to solve complex problems. Nowadays, thanks to technology advancement, enterprises deploy HPCs to process numerous amount of raw data to make data-driven decisions and predictions that will drive revenue for them. In this chapter hardware and software aspects of HPCs are studied and various available platforms and applications are discussed to clarify requirements and challenges of modern HPCs.

3.1 Platform and Underlying Hardware

One important aspect of high performance computing challenges depends on underlying hardware in order to meet time and power constraints. Mainly, three different approaches are used to map desired hardware on silicon to run parallel computations.

1. Application Specific Integrated Circuits(ASIC)
2. Field Programmable Gate Arrays(FPGA)
3. Graghical Processing Unit(GPU)

Nowadays, GPUs and FPGAs are more popular than ASICs mainly due to the costly, laborious and non-recurring design cycles of ASICs. On the other hand, GPU and FPGA platforms have their own advantage and each can cope with the specific type of HPC applications with different performance-per-watt results which will be discussed in the following parts of this work in more details.

3.1.1 GPU

Graphical Processing Unit (GPU) is a dedicated electronic circuit which is designed to perform computations and memory managements in parallel fashion to improve the performance of heavy workloads. In this work NVIDIA GPUs are deployed as target platforms, however, their performance is higher in comparison to FPGAs in the same technology nodes, but performance-per-watt analysis suggests that GPUs have more costly results than Xilinx FPGAs in terms of power consumption. This issue will be discussed extensively in the next chapter using standard OpenCL benchmarks.

NVIDIA's GPUs have already redefined and accelerated High Performance Computing (HPC) capabilities in areas such as seismic processing, biochemistry simulations, weather and climate modeling, signal processing, computational finance, computer aided engineering, computational fluid dynamics, and data analysis [41].

Several architectures, generation after generation, have been developed by NVIDIA in order to improve both performance and overall floating point operation per second (FLOPS) / cost. Table 3.1 lists successive family members of NVIDIA GPUs architecture.

Table 3.1 Release sequence of NVIDIA GPUs micro-architecture

Tesla	90 nm ,80 nm , 55 nm , 40 nm
Fermi	40 nm and 28 nm
Kepler	28 nm
Maxwell	28 nm
Pascal	14 nm and 16 nm
Volta	12 nm

Each generation redefine its predecessor in order to achieve more deeper pipelined hardware with minimum possible resources which can offers more optimized platform to different applications. This work discuss some very interesting techniques used in KEPLER and MAXWELL architecture both in memory hierarchy and processing cores architecture which result in deeper pipelined hardware with respect to their predecessors.

The following new techniques in Kepler increase GPU utilization and parallel programming capabilities[42, 43].

- **Dynamic Parallelism**: Adds the capability for the GPU to generate new work for itself, synchronize on results, and control the scheduling of that work via dedicated, accelerated hardware paths, all without involving the CPU. By providing the flexibility to adapt to the amount and form of parallelism through the course of a program's execution, programmers can expose more varied kinds of parallel work and make the most efficient use the GPU as a computation evolves. This capability allows less-structured, more complex tasks to run easily and effectively, enabling larger portions of an application to run entirely on the GPU. In addition, programs are easier to create, and the CPU is freed for other tasks.

- **Hyper-Q**: Hyper-Q enables multiple CPU cores to launch work on a single GPU simultaneously, thereby dramatically increasing GPU utilization and significantly reducing CPU idle times. Hyper-Q increases the total number of connections (work queues) between the host and the GPU by allowing 32 simultaneous, hardware-managed connections (compared to the single connection available with Fermi). Hyper-Q is a flexible solution that allows separate connections from multiple CORE streams, from multiple Message Passing Interface (MPI) processes, or even from multiple threads within a process. Applications that previously encountered false serialization across tasks, thereby limiting achieved GPU utilization, can be accelerated without changing any existing code.

- **Grid Management Unit**: Enabling Dynamic Parallelism requires an advanced, flexible grid management and dispatch control system. The new Kepler Management Unit (GMU) manages and prioritizes grids to be executed on the GPU. The GMU can pause the dispatch of new grids and queue pending

and suspended grids until they are ready to execute, providing the flexibility to enable powerful runtimes, such as Dynamic Parallelism. The GMU ensures both CPU- and GPU-generated workloads are properly managed and dispatched.

- **NVIDIA GPUDirect™**: NVIDIA GPUDirect™ is a capability that enables GPUs within a single computer, or GPUs in different servers located across a network, to directly exchange data without needing to go to CPU/system memory. The RDMA feature in GPUDirect allows third party devices such as SSDs, NICs, and IB adapters to directly access memory on multiple GPUs within the same system, significantly decreasing the latency of MPI send and receive messages to/from GPU memory. It also reduces demands on system memory bandwidth and frees the GPU DMA engines to used by other CUDA tasks.

- **Streaming Multiprocessor (SMX) Architecture**: Each of the Kepler SMX units feature 192 single-precision CUDA cores, and each core has fully pipelined floating-point and integer arithmetic logic units.

  One of the design goals for the Kepler SMX was to significantly increase the GPU's delivered double precision performance, since double precision arithmetic is at the heart of many HPC applications. Kepler SMX also retains the special function units (SFUs) for fast approximate transcendental operations as in previous-generation GPUs, providing 8x the number of SFUs of the Fermi(figure 3.1).

- **Quad Warp Scheduler**: The SMX schedules threads in groups of 32 parallel threads called warps. Each SMX features four warp schedulers and eight instruction dispatch units, allowing four warps to be issued and executed concurrently. Kepler's quad wrap architecture schedule all four wraps(each has 48 cores) which each can execute two independent(paired) instruction per clock cycle. This means each SMX can execute 8 double precision independent instructions concurrently that results in 108 total instructions for 16 SMX used in GK210 manufactured by NVIDIA™.

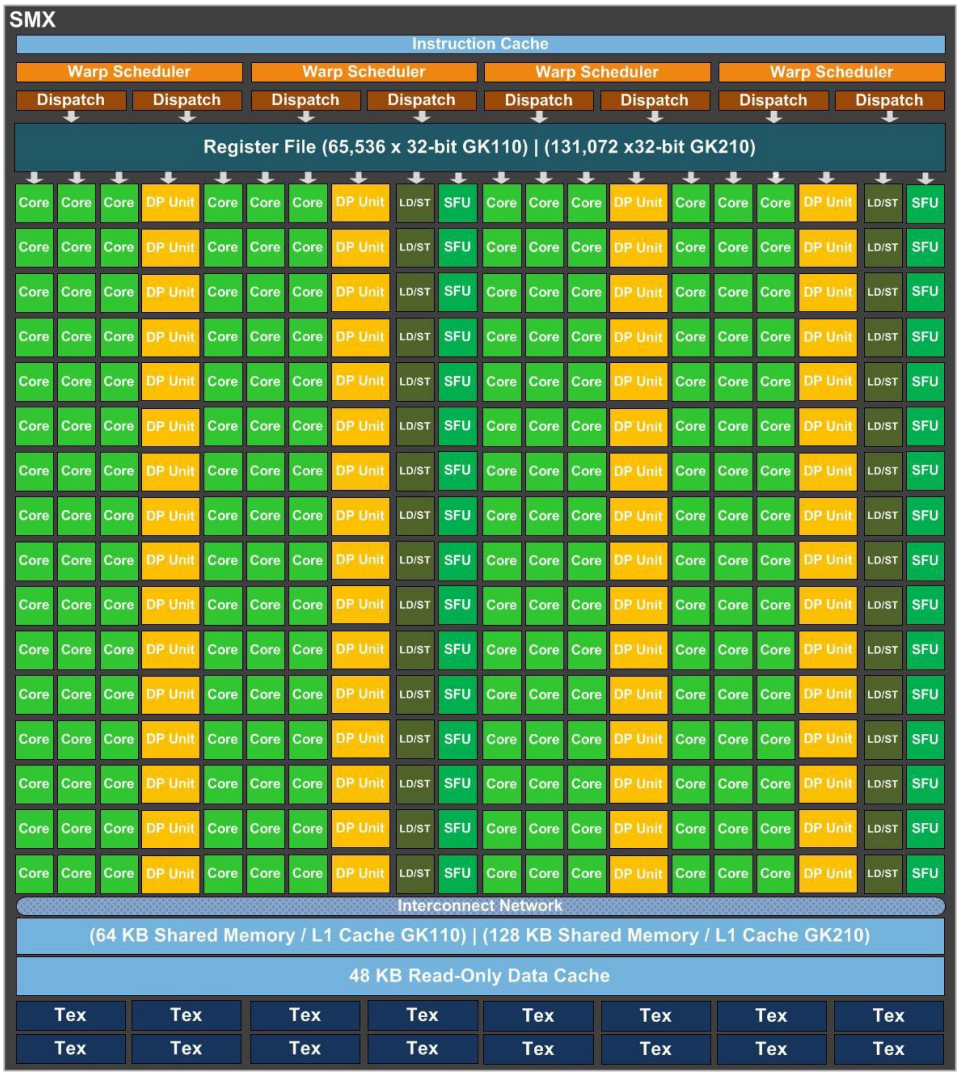


Figure 3.1 SMX: 192 single-precision CUDA cores, 64 double-precision units, 32 special function units (SFU), and 32 load/store units (LD/ST).

To handle massive parallel-architecture of GPU cores an array of fast registers with separate read and write ports are used which is called register file. Size and architecture of register file is paramount to exploit maximum possible parallelism without performance penalty. Nowadays, large register file is used to handle context switching between multi-threads, table3.2 reports L1, L2 and register file size for different GPU devices. The recent GPUs have larger register file than L1, L2 caches which may consume up to 15% of total dynamic power[44].

Table 3.2 NVIDIA GPUs L1, L2 and register file size(Sizes are in KB)

	Architecture	L1 Size	L2 Size	RF size
G80	Tesla	None	None	512S
GT200	Tesla	None	None	1920
GF100	Fermi	48	768	2048
GK110	Kepler	48	1536	3840
GK210	Kepler	48	1536	7680
GM204	Maxwell	48	2048	4096

Kepler's memory hierarchy is organized similarly to Fermi. The Kepler architecture supports a unified memory request path for loads and stores, with an L1 cache per SMX multiprocessor. Kepler GK110 also enables compiler-directed use of an additional new cache for read-only data, Figure 3.2.

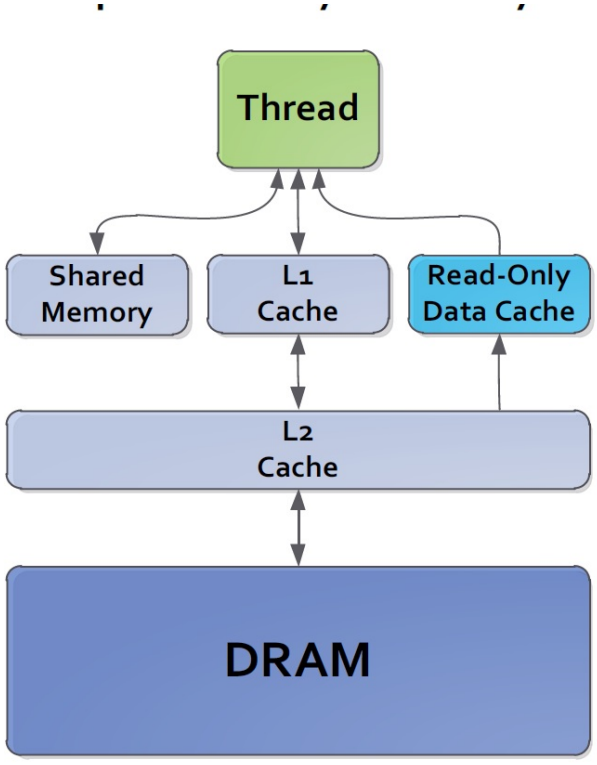


Figure 3.2 Kepler Memory Hierarchy

Maxwell architecture is the successor of Kepler and drive the most high-performance silicon among Fermi, Kepler and Maxwell family which are all using 28 nm transistors. Maxwell design retain all discussed powerful features of its predecessors i.e. Fermi and Kepler architecture along with additional optimization in micro architecture level to obtain higher compute capacity by performing fine-grained parallelism among different section of SMM (Streaming Maxwell's Multiprocessors is shown in figure 3.3).

Maxwell architecture with larger shared memory, higher clock rates and more transistors not only can outperform Kepler architecture, but also it is more energy efficient mainly because of more thoughtful design of streaming multiprocessors. Interestingly, both NVIDIA reports and reports from our experiments in POLITO lab suggest up to 40% performance advantage and twice more energy efficient CUDA cores for MAXWELL architecture with respect to Kepler. Table3.3 reports some important parameters in two GPU devices manufactured with 28 nm transistors using two different architecture[45].

Table 3.3 Comparison of Kepler and Maxwell architecture

GPU	Kepler(GK110)	Maxwell(GM204)
CUDA cores	1536	2048
Base Clock	1006 MHz	1126 MHz
GFLOPs	3090	4612
Shared Memory / SM	48 KB	96 KB
Memory Clock	6008 MHz	7010 MHz
L2 Cache Size	512 KB	2048 KB
TDP	195 Watts	165 Watts
Transistors	3.54 billion	5.2 billion

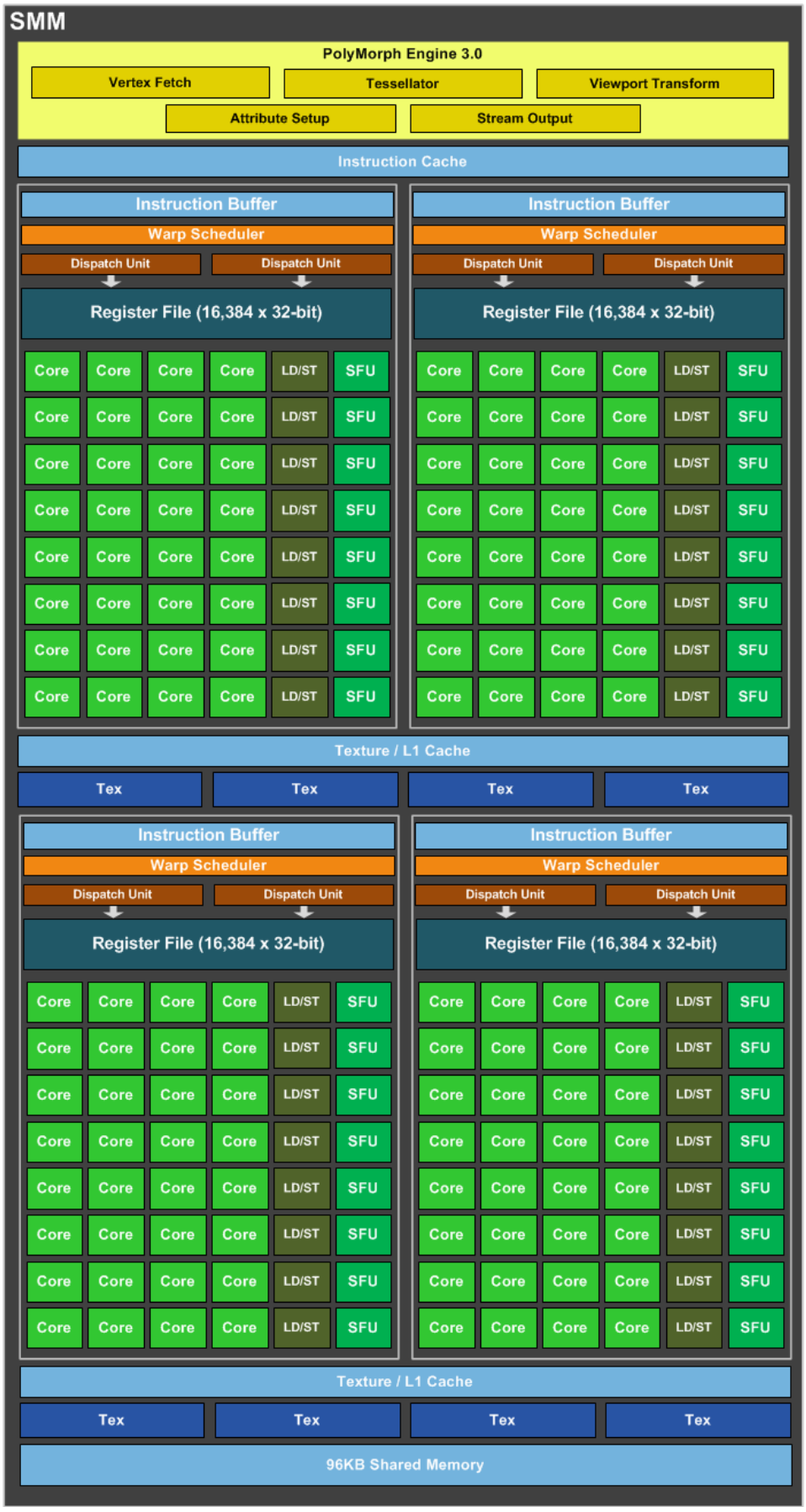


Figure 3.3 Streaming Maxwell's Multiprocessors

3.1.2 High Bandwidth Memory(HBM):

Ever growing number of on-chip cores in modern processors demands high band-width and low power memories to deliver gigabytes of data to many-core processors in power-efficient fashion. HBM is the modern memory[1] with the 3D-stacked of DRAM adopted by JEDEC[2] as an industry standard. The HBM DRAM is designed to operate in lower frequency with respect to GDDR5 while delivering higher performance to system using independent and distributed interfaces called channels. Channels are independently clocked and they can work in parallel without the need of synchronization(figure 3.4). Table 3.4 reports key parameters of HBM and GDDR5 memory, however, GDDR5 uses higher clock rates and voltage, but HBM with larger number of independent interfaces offers higher overall band-width per watt which breaks performance limitation caused by inefficient power consumption of GDDR5 memory subsystems[46].

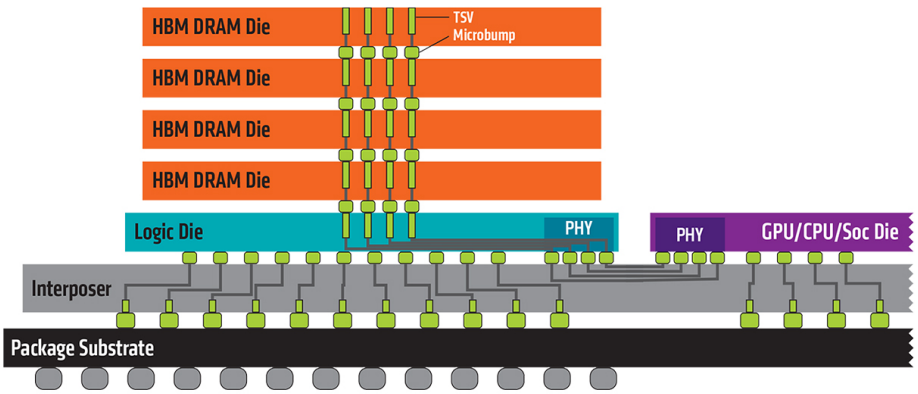


Figure 3.4 Stack of Memory Chips

[1]High Bandwidth Memory has been adopted by JEDEC as an industry standard in October 2013.The second generation, HBM2, was accepted by JEDEC in January 2016.

[2]The JEDEC Solid State Technology Association is an independent semiconductor engineering trade organization and standardization body.

Table 3.4 Comparison of GDDR5 and HBM

Per Package	Bus Width	Clock Speed	Bandwidth	GB/s per watt
HBM	1024-bit	Up to 500MHz	100 GB/s per stack	35
GDDR5	32-bit	Up to 1750MHz	28GB/s	10.66

In addition to performance and power efficiency, HBM has much smaller footprint with respect to GDDR5. As accelerators increasingly demand smaller and more powerful memory, compared to GDDR5, HBM can fit the same amount of memory in 94% less space [47]. Figure 3.5 illustrates how HBM outperform GDDR5 using stacked of memory using shorter and more efficient connection to silicon die using a silicon layer named interposer developed by AMD and Nvidia. Although, using HBM improve performance-per-watt of accelerator and remove possible limitation caused by power budget, but overall cost of chips increase significantly due to modern and challenging deployed techniques of HBM fabrication instead of traditional printed circuit board (PCB) methods.

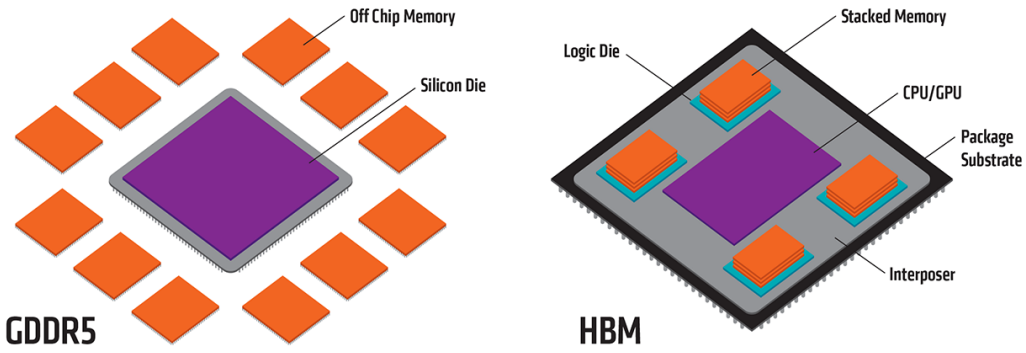


Figure 3.5 HBM(7 mm * 5 mm) vs GDDR5(24 mm * 28 mm) footprint

3.1.3 FPGA

FPGA architecture and possible programming methods are discussed in chapter one. Moreover, as studied in previous section, GPU has dedicated ASIC-like design that can address HPCs applications coming from various domain. In next chapter of this work advantages and advancements of FPGA programming will be discussed in order to perform a scientific and practical analysis on FPGA and GPU devices which is also the main motivation of conducting this PhD program, but before chapter 4, two common computational intensive workloads are discussed algorithmically which will be used to draw analytical results of chapter 4.

3.2 Applications

In the rest of this chapter two different HPC applications are discussed algorithmically to shed lights on high performance computing applications from different angles.

1. **Join Operation:** Join operations are at the core of all relational databases which combine two given arrays based on a given criteria.
2. **Frequency Modulated Continuous Wave (FMCW) RADAR:** FMCW radar which is a high performance and real time application used in automotive industry.

3.2.1 Join Operation

Join operations are at the core of all relational databases. Their performance on CPU-based platforms has been discussed extensively for several decades. This section introduces and illustrates the pseudo-code of the nested loop and sort-merge join approaches and covers previous similar studies.

Related Works and background

This section summarizes related work on implementation of the join operation using GPU and FPGA accelerators, with the main focus on Xilinx FPGAs and high-level synthesis.

The authors of [48] discuss relative performance of the nested loop and sort-merge join algorithms. However, they do not discuss a specific target platform. Their results confirm that the sort-merge join algorithm outperforms the nested loop join algorithm except for small data sizes. In [49] modern multi-core processors are compared via an extensive analysis of their performance executing of sort-merge join and radix-hash join. Their results indicate that only when very large amounts of data are involved sort-merge join has better performance than radix-hash join.

Two different hardware implementations of the bitonic sorting network were presented in [50]. The best performing design, in that case, utilized a single memory port and a streaming permutation network (SPA), thus resulting in a memory and energy optimized implementation on a Xilinx Virtex-7 platform. A significant performance improvement was also achieved in [51] by proper pipelining of different stages of the sorting network. In [52], the Bitonic sort algorithm was compiled for a GPU-based hardware platform by using CUDA, where optimizations were done mainly to reduce the number of global memory accesses and the number of kernel launches.

Even though GPUs and CPUs have been the main platform for query processing, FPGAs have recently gained interest due to the availability of FPGA-based reconfigurable computing [53]. Implementation of database systems on FPGA is now much easier, as a result of the availability of OpenCL-based and C-based design flows.

In [54, 55] and [56] the authors discuss the usage of an OpenCL-based synthesis framework targeting FPGAs that encourages many software developers to use them as acceleration platforms. Although using OpenCL as a high-level synthesis input language is not yet mature and significant hurdles should be addressed to achieve high-quality RTL generation, the design speed offered by the new flow more than overcomes any limitations [56].

Nested Loop join Algorithm:

The nested loop join, illustrated in Algorithm 2, is a straightforward approach to join two relations. Since each loop iteration of this algorithm is completely independent of the others, it offers a huge level of parallelism, but also requires a huge memory bandwidth. This is because the complexity in terms of the number of both of memory reads and writes, and of comparison operations is proportional to the *product* of the sizes of the tables being merged (i.e. $O(n^2)$ if the have the same size). Hence this case is almost ideal in terms of raw parallelism, but is absolutely brutal in

terms of usage of memory bandwidth [33].

Algorithm 2: Nested Loop Join Algorithm

Input: vector A[N] and B[N] with Size of N ,M;

Output: A_out[N] ,B_out[M] and Value_out[N*M]; Output arrays store indices and values of input after join operation respectively

```
1  for each i ∈ {1...M} do
2      for each j ∈ {1...N} do
3          if A[j] and B[i] can be joined (have the same key) then
4              A_out[i*N+j]=j; write current index of A into output
5              B_out[i*N+j]=i; write current index of B into output
6              Value_out[i*N+j]=A[j]; write current value into output
7          end
8          else If the join condition is not met mark output with holes
9          A_out[i*N+j]=-99;
10         B_out[i*N+j]=-99;
11         Value_out[i*N+j]=-99;
12     end
13 end
```

Sort Merge Join Algorithm:

The main idea behind the sort-merge join algorithm is to that sorts each vector before performing join phase. Then the join process of two sorted vectors can be implemented using one single loop with complexity $O(N+M)$, whose execution time is negligible in comparison with sorting (which is close to $O(NlogN)$).

Bitonic sorting is one of the fastest known sorting networks. In general, the term "sorting network" identifies a sorting algorithm where the sequence of comparisons is not data-dependent, thus making it suitable for parallel hardware implementation. A simple example of sorting network is depicted in Fig. 3.6, with five comparators and four inputs. The comparators in a layer can work concurrently (i.e. they can be part of a kernel in OpenCL).

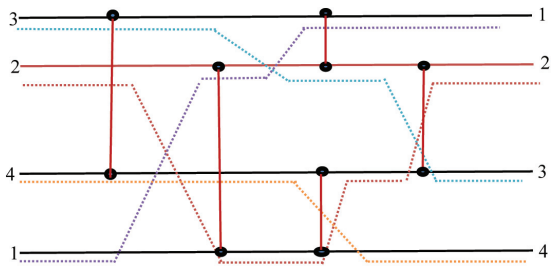


Figure 3.6 Illustration of a simple sorting network

The depth and number of comparators are key parameters to evaluate the performance of a sorting network. The depth of a sorting network is the maximum number of comparators along any path. If all the comparisons in each layer could be done in parallel (i.e. with infinite resources), the depth of the network would be proportional to the total execution time. The bitonic sort network shown in Fig. 3.7, is one of the fastest comparison sorting networks, where the depth is $D(N) = \frac{log_2N.(log_2N+1)}{2}$ and the number of comparators is $C(N) = \frac{N.log_2N.(log_2N+1)}{4}$.

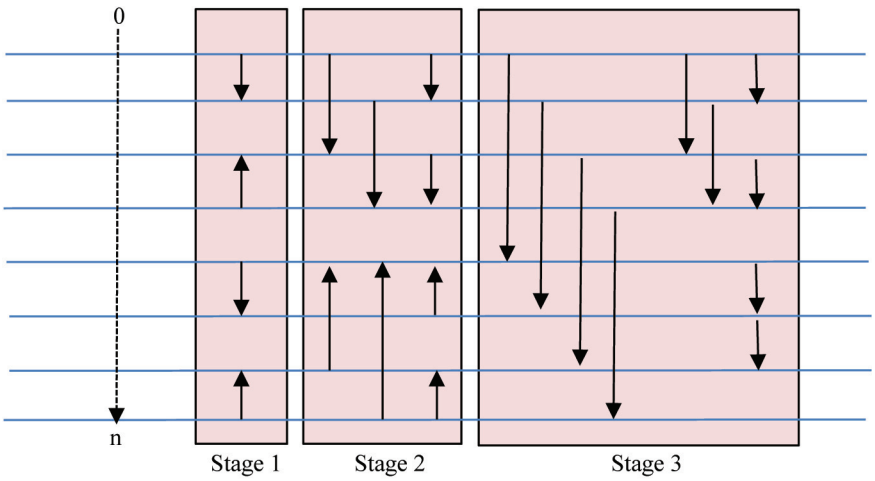


Figure 3.7 Bitonic sort network with 8 inputs (N=8). It operates in 3 stages, it has a depth of 6 (steps) and employs 24 comparators.

Bitonic sorting is a recursive divide-and-conquer algorithm that is based on the notion of bitonic sequence, i.e. a sequence of N elements in which the first K elements are sorted in ascending order, and the last $(N-K)$ elements are sorted in descending order (i.e. the $K-th$ element acts as a divider between two sub-lists, each sorted in a different direction), or some circular shift of such an order.

Bitonic sorting first divides the input into pairs of keys and sorts them into a set of bitonic sequences. It then repeatedly merges and sorts pairs of adjacent bitonic sequences, until the entire sequence is sorted [52].

Algorithm 3 & 4 and 5:
Algorithm 3, executed on the host (i.e. the CPU), iterates the execution of three kernels described in Algorithm 4 to complete bitonic sorting in three phases. In the first phase, the algorithm partially sorts an arbitrary input array to obtain a set of bitonic sequences. In the second and third phases respectively, the algorithm merges bitonic sequences repeatedly until a fully sorted array is produced. All the comparisons (i.e. all the WGs and WIs) in each kernel execution can be performed in parallel and independent of each other. This makes the whole algorithm suitable for parallel implementation.

Algorithm 3: Host Code for Bitonic Sorting execution

Input: A vector of N keys to be sorted;
Output: A sorted vector of the same keys;

```
1  Begin
2  On host:
3  SORTLOCAL(Input, Output);
4  for size = 4 * Work_Group_Size to N do
5      multiply size by 2;
6      for stride= size/2 to stride > 0 do
7          divide stride by 2;
8          if stride >= 2 * Work_Group_Size then
9              MERGE LOCAL(Input, Output, size, stride);
10         end
11         else
12         MERGE GLOBAL(Input, Output, size, stride);
13         end
14     end
15 end
16 End
```

Algorithm 4: Bitonic Sorting Kernels

```
1  On device:
2  Begin
3  function KERNEL1: SORT LOCAL(Input, Output)
4  for local_id = 0 to Work_Group_Size − 1 do
5      copy a block of data from global to local memory with the size of
         work_group;
6      for size = 2 to size < Work_Group_Size do
7          multiply size by 2;
8          for stride = size/2 to stride > 0 do
9              divide stride by 2;
10             perform comparison on each pair and swap them if they are not
                 sorted;
11         end
12     end
13     for stride = Work_Group_Size to stride > 0 do
14         divide stride by 2;
15         pos = 2 ∗ local_id − (local_id&(stride − 1));
16         compare and sort each pair;
17     end
18     write back sorted array to global memory with the size of work_group;
19 end
20 function KERNEL2: MERGE LOCAL(Input, Output, size, stride)
21 for local_id = 0 to Work_Group_Size − 1 do
22     read one pair in each Work Item;
23     perform comparison on each pair and swap them if they are not sorted;
24     write back sorted pair to the global memory;
25 end
26 function KERNEL3: MERGE GLOBAL(Input, Output, size, stride)
27 declare and initialize a private variable global_stride;
28 for local_id = 0 to Work_Group_Size − 1 do
29     copy a block of data from global to local memory with the size of
         work_group;
30     for stride = global_stride to stride > 0 do
31         divide stride by 2;
32         perform comparison on each pair and swap them if they are not
             sorted;
33     end
34 end
35 End
```

Join Algorithm:

Algorithm 5: Join Algorithm for Sorted Relations

Input: A[N] and B[M] are two sorted vector
A_index[N], B_index[M] contain indices of A and B before being sorted;

Output: A_out[N+M] ,B_out[N+M] ,Value_out[N+M]; If join condition is met output arrays store indices and values of inputs respectively

```
1  Begin
2  Function Sort_Join
3  Initialize i,j,k, t_tmp = 0; i,j are indices of inputs , k is the outputs index and
     j_tmp is used to check for successive join between array B elements and the
     same element of A
4  while (i < N and j < M)
5  if A[i]>B[j] then
6      j++;
7  end
8  else
9  if A[i]<B[j] then
10     i++;
11 end
12 else  If the join condition is met write indices and values of inputs into output
13 A_out[k]=A_index[i];
14 B_out[k]=B_index[j];
15 Value_out[k]= A[i];
16 j_tmp=j+1;
17 k++;
18 while(A[i]==B[j_tmp])
19 A_out[k]=A_index[i];
20 B_out[k]=B_index[j_tmp];
21 Value_out[k]= A[i];
22 j_tmp++;
23 k++;
24 end
25 i++;
26 end
27 end
28 end function
29 End
```

Finally, algorithm 5 performs the join algorithm on two sorted vectors within a single linear complexity loop. CPU can handle join operation with single loop complexity without the need of accelerator, join operations are fundamental in relational data base management.

When it comes to deep hardware accelerator, FPGAs are powerful devices and are able to run fully pipelined stages, moreover, each stage is parallelized locally to decrease latency and improve overall performance using discussed techniques in chapter two and will be reported analytically in next chapter of this work using standard OpenCL models targeting FPGA an GPU devices. The depth and number of comparators of sorting network grow with input size,figures 3.8 and 3.9 plot their behavior.

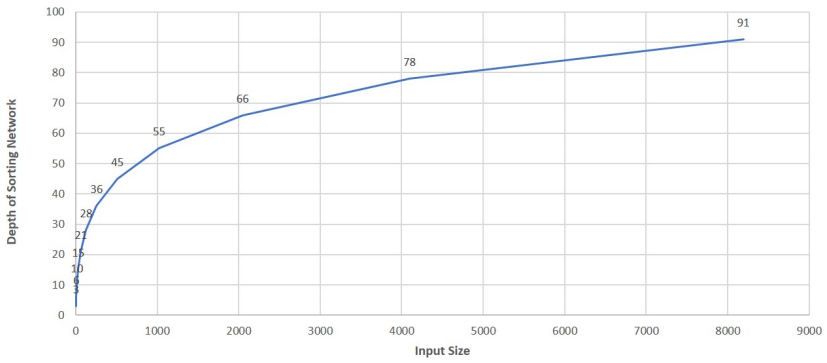


Figure 3.8 Depth of Sorting network : $D(N) = \frac{log_2N.(log_2N+1)}{2}$

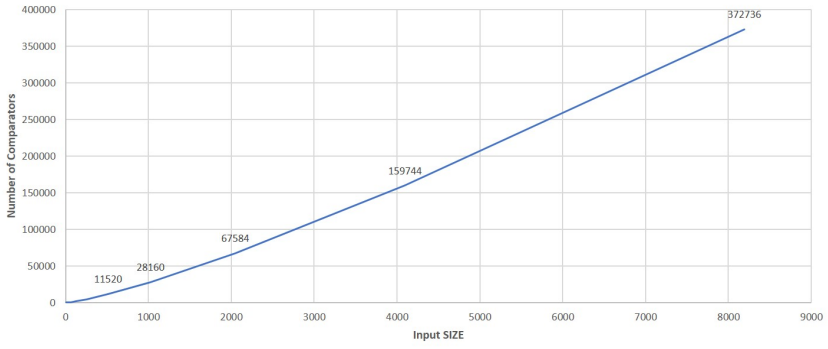


Figure 3.9 Number of Comparators : $C(N) = \frac{N.log_2N.(log_2N+1)}{4}$

3.2.2 Frequency Modulated Continuous Wave (FMCW) Radar

The idea of detecting objects based on Frequency Modulated Continuous Wave (FMCW) is based on transmitting signals with known linear varying frequency and receiving the reflection of signal from surrounding objects. As shown in figure 3.10, during the round trip of continuous wave, received frequency from moving target will shift horizontally and vertically due to Doppler effect which can be translated to distance and velocity off surrounding object using following formulas.

$$Range = (\ C\ /\ (4 * Frequency\ Ramp\ Rate)) * (f_1 - f_2)$$

$$Velocity = (\ C\ /\ (4 * Carrier\ Frequency)) * (f_1 + f_2)$$

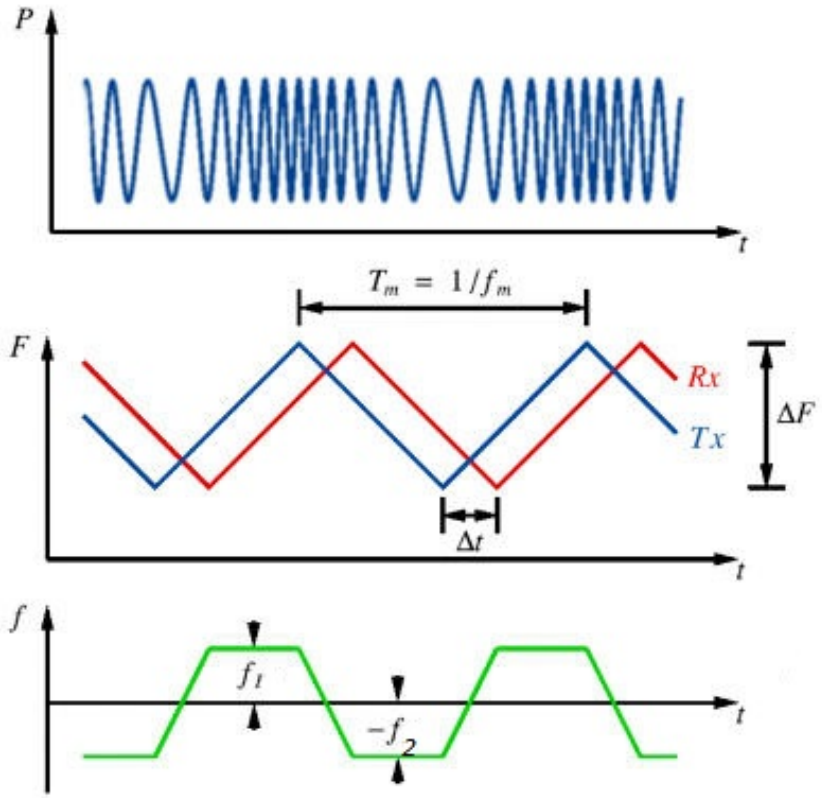


Figure 3.10 FMCW Radar signal analysis

DSP algorithm of FMCW unit of radar is composed of four main sequential sub-algorithms described briefly below (Figure 3.11):

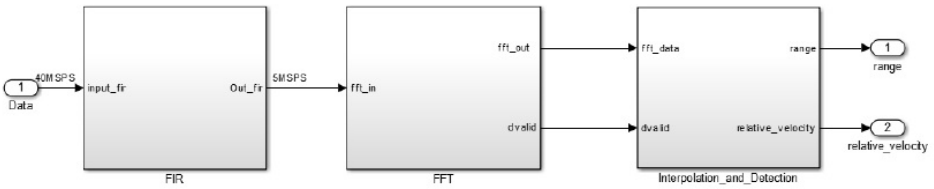


Figure 3.11 Simulink Top model of FMCW

1. FIR filter and Decimation unit improve Effective Number Of Bits (ENOB), in other words, it leverages more efficient signal for next block by lowering sample rate of computation. In fact, decimation is a trade-off between accuracy and computation cost. Decimation factor in this case is 8 that lowers sampling frequency from 40 Mega Sample Per Second (MSPS) to 8 MSPS

2. Fast Fourier Transform (FFT) converts signal from time domain to frequency domain in order to extract information from data signal.

3. Interpolation and peak detection is necessary to detect maximum frequency with acceptable accuracy, as shown in figure 3.12.

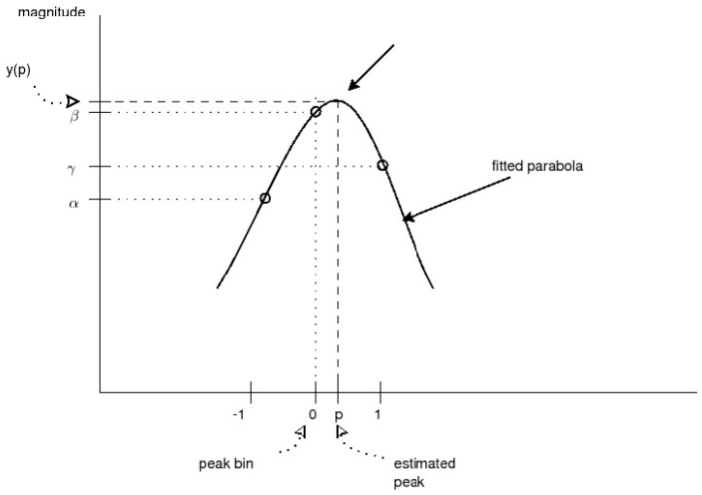


Figure 3.12 Estimation improvement by Interpolation

4. Finally, mathematical formulas of FMCW radar are modeled in Simulink to estimate range and velocity of surrounding objects.

All four sub-algorithms are modeled and verified using Simulink, additionally, both VHDL and C based models are automatically generated using Simulink code generators. Table 3.5 reports computation accuracy of discussed DSP algorithm for three different test cases, table 3.6 reports specification of discussed radar. In this experiment, we modeled a vehicle with 50 Km/hr speed which is equipped with FMCW radar and detects three objects distanced in 30 m moving with three different velocities.

Table 3.5 Computation Accuracy of FMCW RADAR

Test Signal	Range Accuracy (%)	Velocity Accuracy(%)
R=30 m & V= -50 Km/hr	99.8 %	95%
R30 m & V = 0 Km/hr	99.9 %	97%
R=30 m & V = +100 Km/hr	99.8 %	95%

Table 3.6 Specification of Modeled FMCW Radar with FFT of 2048

Carrier Frequency	Frequency Ramp Rate	Maximum Range	Minimum Range
77 GHz	1 GHz/s	300 m	1 m

Generated C code from DSP algorithm is verified using the test bench of top model, Vivado HLS with integrated Eclipse environment enables designers to debug, run and verify C based model before generation of RTL, moreover, Vivado HLS estimates total required number of clock cycles for each sub-function and region of C model after high-level synthesis. Based on this estimation 99% of total required clock cycles for single initiation of FMCW algorithm belongs to FFT function. This motivates us to parallelize FFT model and substitute sequential C code with parallel SystemC version of FFT based on the principles described in the next section.

FFT Algorithm:

The Fourier Transform is widely used in signal processing to transform signals from the time domain to the frequency domain and vice-versa. The Fourier Transform that operates on discrete data is called Discrete Fourier Transform (DFT). The Fast Fourier Transform (FFT) is one of the most famous and widely used algorithms to calculate the DFT and its inverse. The FFT algorithm exploits the symmetry of the calculation and the re-use of already performed calculations to reduce the computation complexity from N^2 to $NLog_2N$ for a DFT that is computed on N samples. The FFT algorithm is selected as case study because it can be represented using different non-trivial signal flow graphs (SFG) and finds application in many signal processing areas. These various signal flow graph representations are beneficial for targeting different application domains with different performance requirements under different constraints.

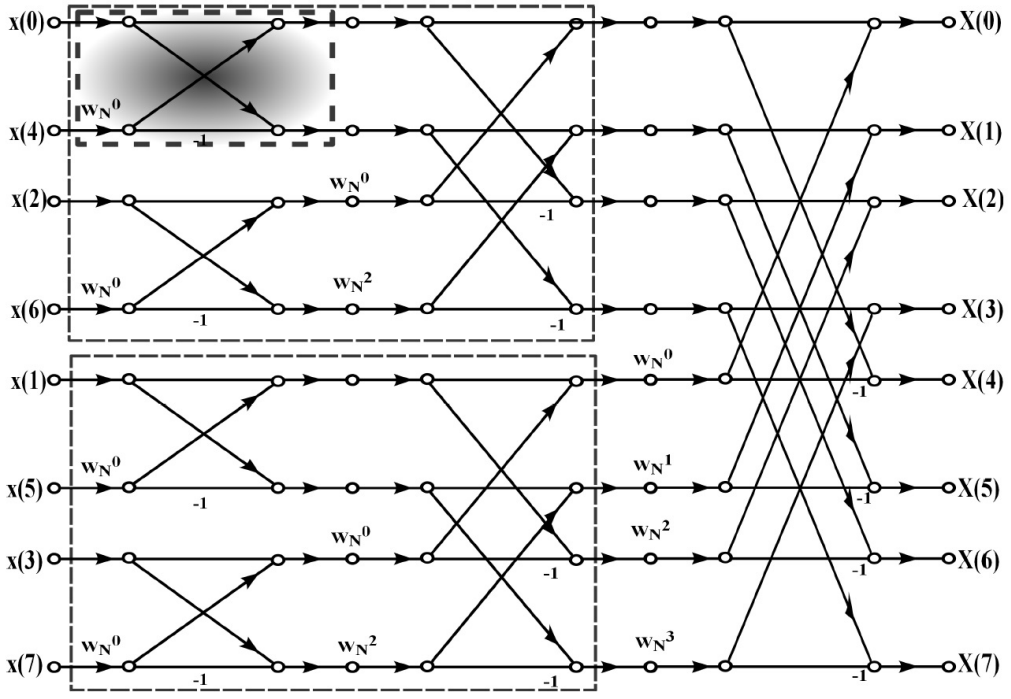


Figure 3.13 Signal flow graph for radix-2, 8-point in place FFT computations

Figure 3.13 shows a signal flow graph (SFG) for computing an FFT with 8 samples. The SFG represents the fully unrolled computations and data dependencies

(and thus the full available parallelism) implied by the C code structure used by RTW for a software oriented FFT implementation. In this SFG each node represents a complex operator and each arrow represents a complex value. It is called radix-2 FFT since its basic unit, called butterfly and marked by the dotted box at the top left of the figure, consumes two input samples to produce two output samples. Constants marked as W_0^N , W_1^N , W_2^N and W_3^N are complex exponentials, known as twiddle factors. Inputs x(0), x(1), . . . x(7) are the complex time domain samples of the signal to be transformed and outputs X(0), X(1), . . .X(7) are the complex values of the frequency spectrum of the signal. Each butterfly represents the multiplication of twiddle factors by input samples and then one addition and subtraction to calculate outputs.

The signal flow graph in Figure 3.13 is called in-place FFT because every butterfly can write outputs to the same memory from where it has read the inputs. Such a representation is useful for implementing a resource shared FFT with relatively low throughput requirements, targeting low power applications with limited on chip memory size and bandwidth. But this kind of signal flow graph is not well suited when throughput requirements are high and either a pipelined implementation or a fully unrolled register-based (rather than memory-based) implementation is required. For example, let us assume that in order to increase throughput we unroll the inner loop that performs butterflies in a stage (a column of Figure3.13), and that stage inputs are mapped to registers. After performing a butterfly computation, the inputs for the next butterflies mapped to the same multiplier/adder/subtractor resources will come from signal flow graph positions that are different from the first stage, which in hardware will imply high multiplexing cost and hence will not be efficient. Similarly, some tools and memory architectures may not efficiently support pipelining of loops in which computations read and write from the same memory, due to the need to use multi-ported memories. This, on the other hand, would be easy in software, for which the graph in Figure 3.13. works best.

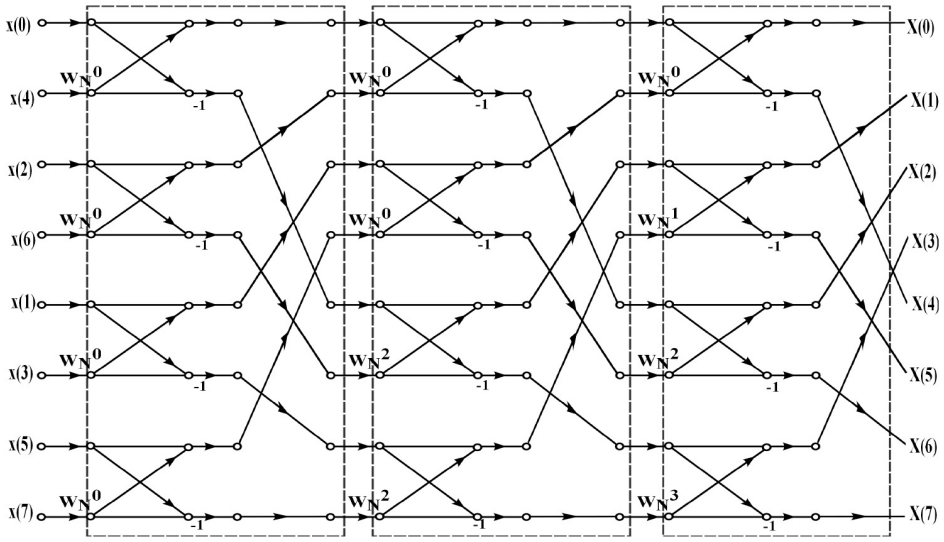


Figure 3.14 Signal flow graph for radix-2, 8-point FFT computations

Figure3.14 shows another signal flow graph for FFT, which also can be represented in C in the form of nested loops, but is much more flexible than the one in Figure 3.13 to derive many possible implementations using HLS. In particular, it can be mapped to a register-based unrolled implementation. The advantage of such an FFT representation with respect to Figure 3.13 is that the interconnection network between the stages is the same for all the stages, which results in less multiplexing cost when a stage is partially or fully unrolled and subsequent stages are implemented by iteration. Even when a memory-based implementation with more aggressive resource sharing and lower throughput is required, still the signal flow graph in Figure 3.14 is more flexible. This is because one can always map inputs and outputs of a butterfly to two different memories, while still allowing partial unrolling depending on the memory read/write bandwidth. The signal flow graph in Figure 3.14 can even be mapped to a single on-chip memory implementation by utilizing the memory merging capabilities offered by HLS tools, which allows one to map two different memories of different lengths and widths (arrays in C) to a single aggregate memory. In our FFT HLS-IP we used the SFG in Figure 3.14, because it can offer broader design space exploration as compared to Figure 3.13, which corresponds to the default software implementation from Simulink RTW. Moreover,in chapter 4 resource utilization of our fully pipelined FFT HLS-IP and HDL code of Simulink FFT are presented and compared.

Chapter 4

Implementation and Performance-per-Watt Analysis of HPC Applications on FPGA-GPU Platforms

In this chapter, first we demonstrate the capabilities of high level synthesis methodology by comparing performance and resource utilization of automatically generated HDL and C codes from verified Simulink model targeting Xilinx FPGAs[15][1]. In fact, we generate HDL-based and C-based IPs from top model to program FPGAs using Xilinx synthesis tools, namely VIVADO and VIVADO HLS.

In the next stage of this experiment, performance and power analysis on FPGA and GPU platforms are conducted using different OpenCL benchmarks, this helps to draw analytical conclusion from our experiments for each device using standard OpenCL benchmark. Moreover, extensive optimizations are performed to enhance the quality of generated RTL using SDAccel which will be discussed in this chapter.

[1]The procedures of automatic code generation using Embedded coder (C/C++) and HDL coder (VHDL/Verilog) are discussed in chapter three.

4.1 FFT-based Digital Signal Processing Unit of Radar

As discussed extensively in chapter three, a radar is an electronic device that is used for estimating different parameters (e.g. speed, direction and position) related to the movement of an object. Radars typically find uses in military and commercial applications. In particular, they are an essential component of assisted driving applications in automotive electronics, e.g. parking assistance, lane departure warning and collision avoidance. In this case study we experimented with the Digital Signal Processing (DSP) unit of a radar for automotive applications based on the Frequency Modulated Continuous Wave (FMCW) technique. The FMCW radar transmits a frequency-modulated signal that will be reflected from a target object. The reflected signal is captured and different parameters, such as the time of flight and Doppler shift, are estimated as shown in figure 4.1. Then these can be translated into the distance and the velocity of the object.

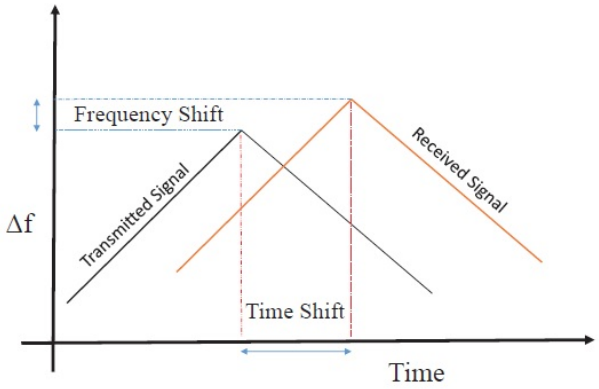


Figure 4.1 Operation of FMCW Doppler Radar

In our experiments we modeled using Simulink the digital signal processing unit of this radar, as shown in figure 4.2. The most expensive block is a high precision 2048 sample FFT. In this experiment we targeted the implementation to a Kintex-7 FPGA (xc7k160tfbg484-3) from Xilinx. Table 4.1 shows the synthesis results only for the FFT, using the performance requirement of the full radar application. It illustrates that our HLS-IP, synthesized using Vivado HLS, uses similar resources when compared to the optimized RTL implementation from HDL coder, for the same real-time throughput constraints. Note that the LUT cost obtained via HLS uses more LUTs because the resulting RTL is less efficient for exploiting the DSP48

units. This is something that we will consider for the future, e.g. by automatically generating FPGAs specific mapping directives for the HLS tool.

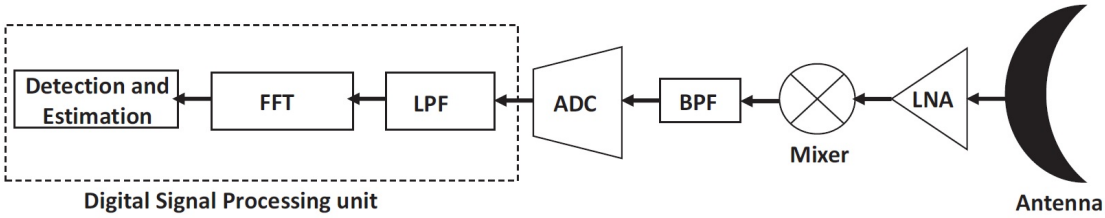


Figure 4.2 FMCW Radar receiver architecture

Table 4.1 FFT Implementation for Radar DSP Unit HLS-IP vs. HDL Coder

Synthesis Tool	DSP 48	LUT	FF	Memory Blocks
Viado HLS (SystemC IP)	80	9069	7015	22
Vivado(RTL IP)	72	7744	11524	25

Table 4.2 Comparison of implementation cost for different radar blocks using HLS

Radar Block	DSP 48	LUT	FF	Memory Blocks
FFT	80	9069	7015	22
LPF	6	901	1529	0
Detection& Estimation	24	11858	4927	0

Table 4.3 Full Radar DSP Unit Implementation HLS vs. HDL Coder

Synthesis Tool	DSP 48	LUT	FF	Memory Blocks
Viado HLS (SystemC IP)	110	21828	13471	22
Vivado(RTL IP)	288	13268	11878	25

The complete synthesis result for radar DSP front-end are reported in Table 4.2. As shown in Table 4.3, the results obtained using HLS-IP and high level synthesis are very comparable with the results obtained using HDL Coder. The HDL Coder based solution uses many more DSP48 slices than results of HLS-IP,which again uses many more LUTs. The other blocks are synthesized starting from the automatically generated C-code produced by Embedded Real Time Coder.

FMCW radar is a real-time application which demands fully pipelined hardware to perform required computation within the time constraint. Figure 4.3 illustrates time diagram of synthesized model from behavioral description. Each output frame between different interval is equal to FFT length (2048 samples, Clock Period = 10 ns). Figure 4.4 shows block diagram of streaming FFT model of Simulink which provides output to next block in a streaming fashion after initial latency caused by buffering and initialization of FFT algorithm.

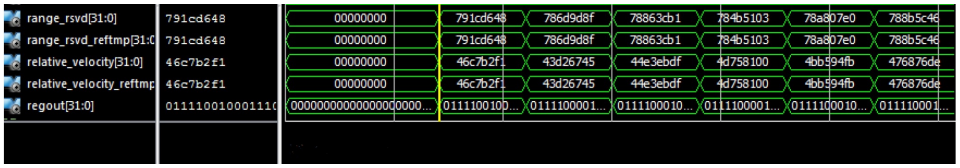


Figure 4.3 RTL simulation of FMCW model generated in ISE Simulator (ISim)

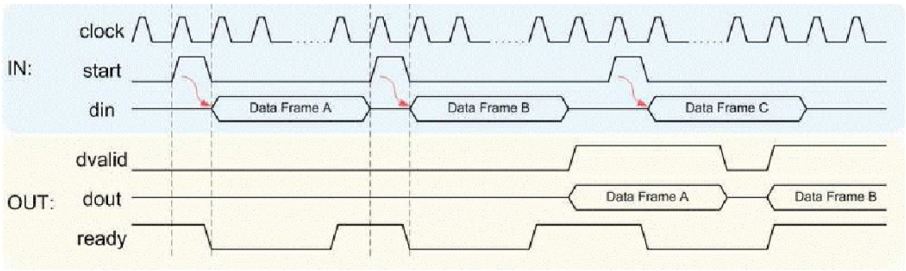


Figure 4.4 Timing diagram of streaming Simulink FFT from HDL library

With the aim of concluding this part of the work in a more comprehensive way, the implementation results of the streaming FFT using DSP Builder which is an alternative MBD tool for Intel FPGAs are discussed as follows [58].

Intel tools such as DSP Builder, enable designers to complete a software-based design flow while targeting FPGAs. DSP Builder for Intel FPGAs eases hardware implementation of DSP functions, provides a top-level verification tool to the system

engineer who is not necessarily familiar with HDL design flow, and allows the system engineer to implement DSP functions in FPGAs without learning HDL. DSP Builder for Intel FPGAs provides an interface from Simulink directly to the FPGA hardware [57].

Figure 4.5 illustrates a numerical comparison between three different implementations of streaming FFT, Figure 4.4 presents the expected timing digram for high throughput FFT. In [58] DSP builder from Altera targets Cyclone V FPGAs which offer the lowest system cost and power for wide range of applications. However, Altera and Xilinx FPGAs architecture are different from each other, in all three approaches 2048 point streaming FFT fits neatly on Cyclone V and Kintex7 which provide the best price/energy at 28 nm devices. **Please note that FFT implementation using HLS methodology is based on the customized SystemC IP discussed in chapter three and [15], moreover, DSE flow described in chapter two is used to obtain the best RTL solution in terms of performance per watt**.

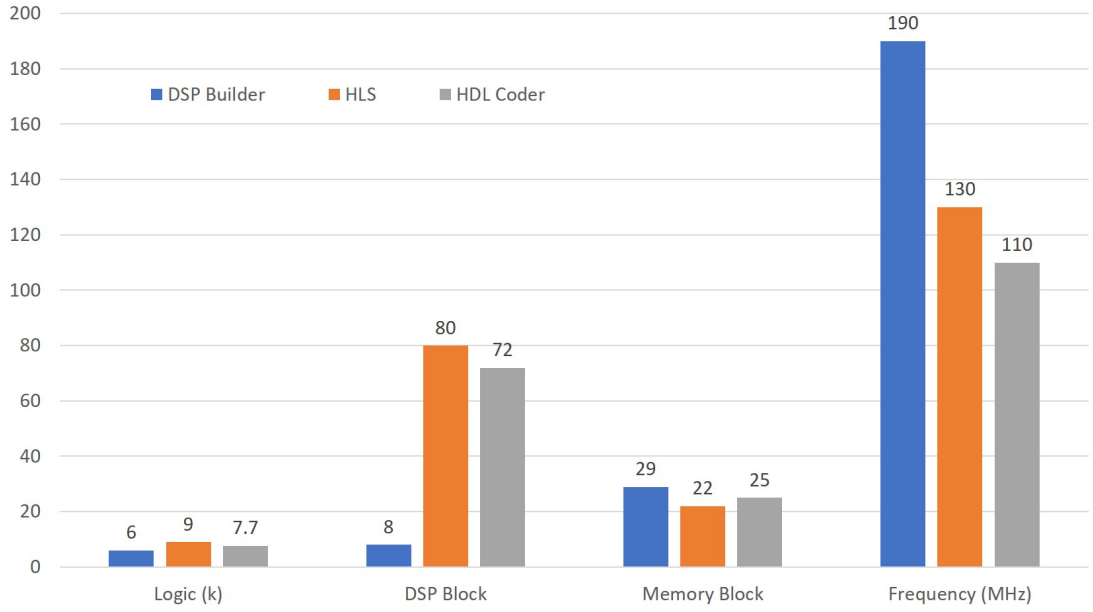


Figure 4.5 Comparison of synthesis result of streaming FFT IP using three different approaches

The conducted experiments indicate that HDL based hardware design delivers the most optimized solution in terms of energy efficiency, thanks to low level pro-

gramming which enables to exploit intrinsic parallel architecture of FPGA in the most sensible way both at macro and micro architectural level . Despite HDL loyalty to underlying hardware, RTL level offers smaller design space with respect to HLS based FPGA design. The HLS can generate high performance hardware with much wider available design space. Figure 4.6, 4.7 and 4.8 demonstrates a comparison of design space exploration of HDL and HLS for different FPGA resources. In all three cases, HLS provides more solutions addressing various application requirements.

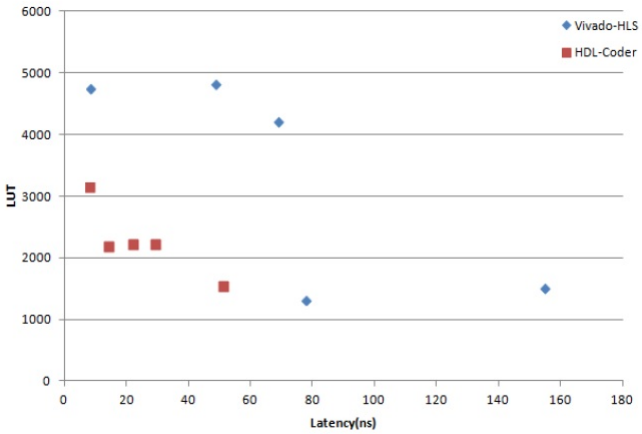


Figure 4.6 HDL and C based design space exploration of FIR subsystem (LUT Utilization)

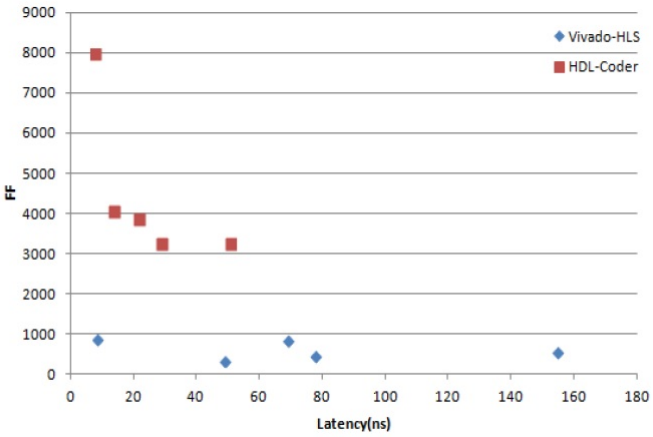


Figure 4.7 HDL and C based design space exploration of FIR subsystem (FF Utilization)

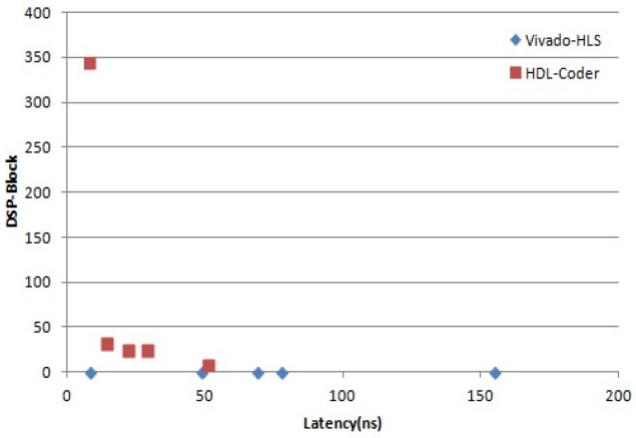


Figure 4.8 HDL and C based design space exploration of FIR subsystem (DSP Utilization)

Next section presents performance comparison between FPGAs and high-end GPUs in terms of execution time, energy and power consumption. The algorithms have been modeled in OpenCL for both GPU and FPGA implementation. However, programming approach is totally different with respect to radar implementation, but using OpenCL provides us with more optimization options in comparison to model based design which is a more software development approach and generation of high quality RTL is only possible for limited set of solutions. Additionally OpenCL enables developers to target GPU/CPU and FPGA devices using same source code that also promises programming heterogeneous platforms.

4.2 Implementation of a Performance Optimized Database Join Operation on FPGA-GPU Platforms Using OpenCL

This section of work presents two implementations of the join operation between two database tables, i.e. the creation of a single merged table containing only elements with the same primary key. One of them is ultra-parallel, based on two nested loops which simply apply the join definition to unsorted tables. The other uses a fast bitonic sort algorithm, with lower complexity,followed by a linear join of sorted tables. Note that sorting is very memory bandwidth-intensive. So this application is a sort of worst case when comparing GPU and FPGA platforms.

4.2.1 Optimization of OpenCL models for FPGAs

SDAccel provides designers with an extensive set of OpenCL attributes that allows a designer to fully control the micro-architecture of synthesized RTL. Optimizations are performed in two phases, micro-architecture and macro-architecture optimization. In our first micro-architectural optimization for both test cases, each kernel is optimized only with respect to its internal structure, by using (1) work item pipelining, (2) loop unrolling and (3) array partitioning. In the next stage of optimization, multiple WGs of each kernel are instantiated on an FPGA, each with its own global memory access port for each OpenCL kernel argument, in order to fully utilize the off-chip memory band-width and increase transfer efficiency.

On-Chip Memory Architecture:

Thoughtful memory architecture of utilized on-chip memory is essential to improve DDR band-with utilization, figure 4.11 illustrates top view of generated hardware by SDAccel, m_axi_interconect_M00_AXI is implemented by memory blocks and LUTs to use all available DDR memory ports. In fact, memory hierarchy of SDAccel region can be presented by figure 4.9 that increase overall performance thanks to more pipelined hardware.

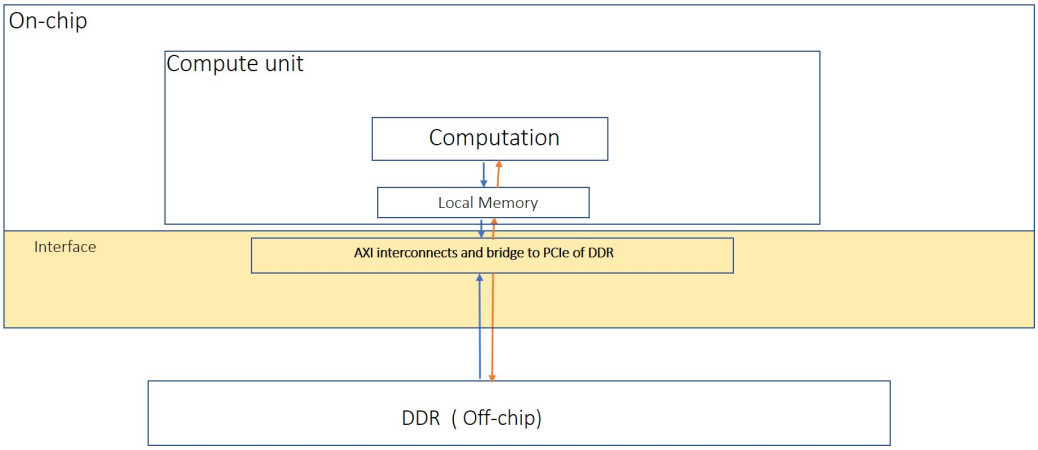


Figure 4.9 Memory model of single compute unit

Fine-grained parallelism of on-chip resources for each compute unit is necessary to increase overall performance. Instructions within the loops, described algorithmically in chapter 3, are unrolled completely and partially. Moreover, local memory of each instantiation of WGs is both partially and completely partitioned to achieve multiple solutions for different constraints. As it is discussed in Appendix A optimum solution can be selected by considering key parameters of each application that ensures better performance-per-watt results for FPGAs in comparison to fixed architecture hardware, e.g. GPUs. Figure 4.10 shows parallel architecture of reconfigurable region of FPGA. Loop unrolling, memory partitioning and reshaping factors provide the possibility of exploring the design space to find the solution with highest performance-per-watt parameter that can be used to generate multi-core RTL with low power consumption.

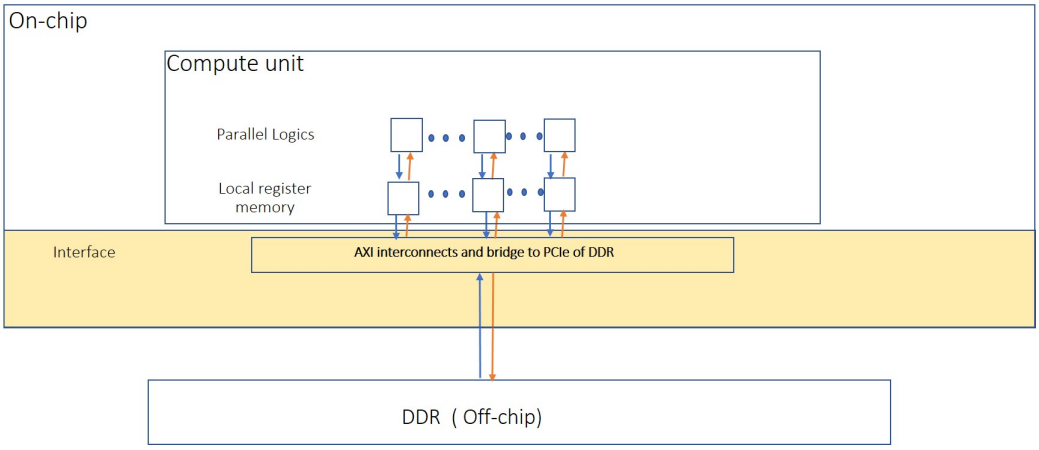


Figure 4.10 Fine-grained memory model of compute unit

Figure 4.11 is the block diagram of SDAccel generated hardware with multiple compute units from OpenCL model using proper optimization directives to address time and power constraints of the design. On the top the master-bridge is connected to available off-chip memory using a PCIE interface. AXI interface is an IP generated by Vivado to connect kernels and bridges. K1, K2 and K3 are multiple instantiations of WGs of same kernel, all executing in parallel. In this figure, each kernel has five global array arguments mapped to global memory.

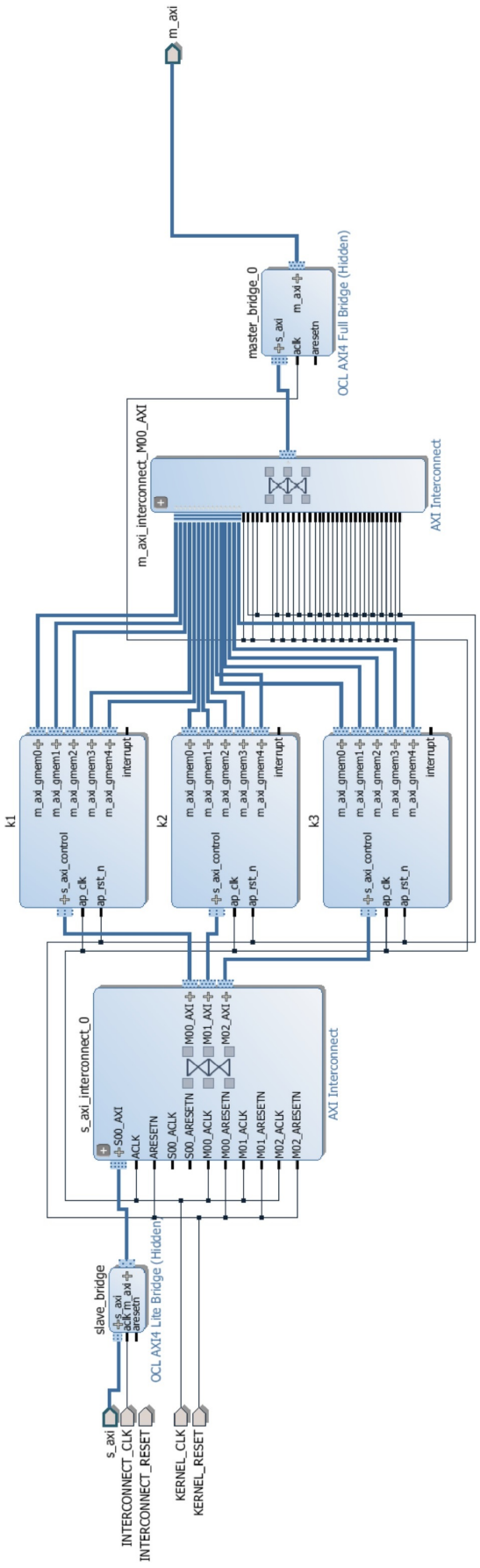


Figure 4.11 Top-level block diagram with 3 OpenCl kernel instances, generated by SDAccel.

4.2.2 Power Analysis

Measurement and analysis of power consumption is necessary to design and select optimum hardware accelerator for HPC applications. Both NVIDIA and Xilinx provide users with profiling tools which can estimate power consumption, memory utilization and the temperature of target device. This section briefly covers these capabilities, additionally, further information on NVIDIA and Xilinx analysis tools can be reached in [59, 21].

NVSMI:

nvidia-smi (also NVSMI) provides monitoring and management capabilities for each of NVIDIA's Tesla, Quadro, GRID and GeForce devices from Fermi and higher architecture families. GeForce Titan series devices are supported for most functions with very limited information provided for the remainder of the Geforce brand.

The "nvidia-smi dmon" command-line is used to monitor one or more GPUs (up to 4 devices) plugged into the system.This tool allows the user to see one line of monitoring data per monitoring cycle. The output is in concise format and easy to interpret in interactive mode. Figure 4.12 shows the output of nvidia-smi dmon command line.

nvidia-smi dmon -i 0(for GTX)/1(for K4200)

In this experiment, our target platform has two different GPU devices specified in table 4.4 which each can be monitored by choosing related GPU idx. NVSMI runs in the background to monitor default metrics for each device under natural enumeration (starting with GPU index 0) at a frequency of 10 second that demands large enough data to keep GPU cores busy for at least one monitoring cycle.

```
^Copencl@mizar:~$ nvidia-smi dmon -i 1
# gpu    pwr  temp    sm   mem   enc   dec  mclk  pclk
# Idx      W     C     %     %     %     %   MHz   MHz
    1     21    49     0     0     0     0   324   324
    1     21    49     0     0     0     0   324   324
    1     21    49     0     0     0     0   324   324
    1    102    54   100    64     0     0  2700   888
    1    106    55   100    64     0     0  2700   888
    1    107    56   100    64     0     0  2700   888
    1    107    57   100    64     0     0  2700   888
    1     73    55   100    39     0     0  2700   888
    1     42    54     0     0     0     0  2700   888
    1     42    54     0     0     0     0  2700   888
    1     31    53     0     0     0     0   810   324
    1     24    53     0     0     0     0   810   324
    1     24    53     0     0     0     0   810   324
```

Figure 4.12 Output of nvidia-smi dmon command line

Vivado Power Analysis:

The Vivado power analysis feature performs power estimation through all stages of the flow: post-synthesis, post-placement, and post-routing. It is most accurate at post-route because it can read the exact logic and routing resources from the implemented design. Figure 4.13 presents the Summary power report and the different views of your design that you can navigate: by clock domain, by type of resource, and by design hierarchy. Within the Vivado Integrated Design Environment (IDE) you can adjust environment settings and design activity so you can evaluate how to reduce your design supply and thermal power consumption. You can also cross-probe into the design from the power report, which aids in identifying and evaluating high power consuming hierarchy/resources used in the design.

Total on-chip power is the aggregate of dynamic and static power described in the following lines.

Static Power:

Device static power is the power from transistor leakage on all connected voltage rails and the circuits required for the FPGA to operate normally, post configuration. This is normally measured by programing a blank bitstream into the device. Device static power is a function of process, voltage, and temperature. This represents the steady state, intrinsic leakage in the device.

Dynamic Power:

This power is instantaneous and varies at each clock cycle. It depends on voltage levels and logic and routing resources used. This also includes static current from I/O terminations, clock managers, and other circuits that need power when used. It does not include power supplied to off-chip devices.

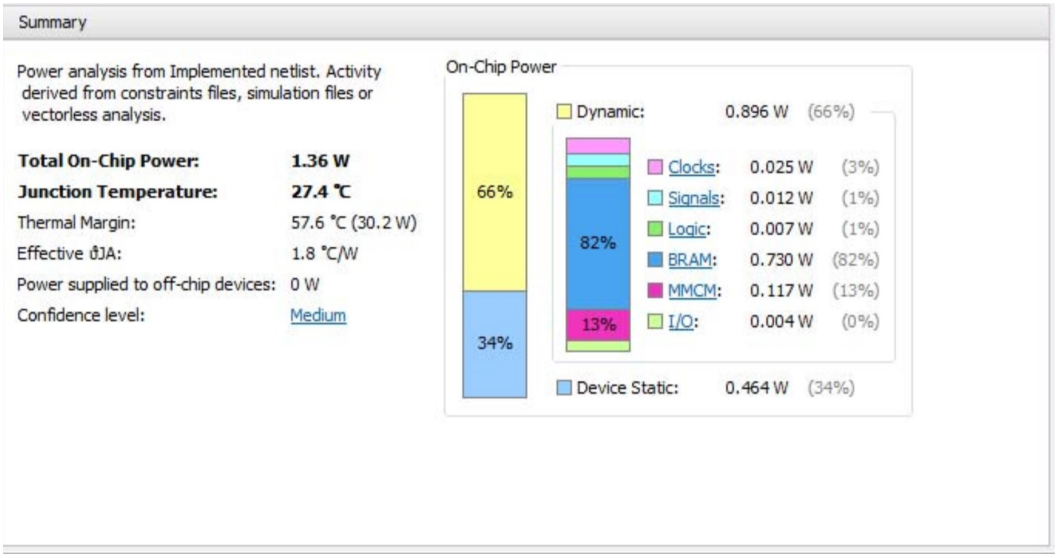


Figure 4.13 Vivado Power Analysis

4.2.3 Performance-per-watt Analysis

This section presents performance-per-watt analysis of target devices in this work using join operations as standard benchmarks. Table 4.4 reports the specification of GPUs [60] and FPGAs used in this study. Although the relative performance of each device can vary from one test case to another, the GTX960 often outperforms the K4200 in our experiments. The higher number of cores and higher memory bandwidth of K4200 are not as effective as one could hope, most likely because of higher core speed (37%) and the use of a second generation Maxwell architecture, with a very large cache, for GTX960.

Table 4.4 Specification of tested Platforms

Params/Devices	**GTX960**	**K4200**	**Virtex7 Series**	**VU440**
Architecture	Maxwell GM206	Kepler GK104	Virtex7	Virtex US
Process	28nm	28nm	28nm	20nm
Cuda Cores	1024	1344	-	-
Core Speed	1127 MHz	706 MHz	-	-
Memory Interface	GDDR5	GDDR5	DDR3	DDR4&HBM
Memory Bandwidth	112.2GB/sec	172.8GB/sec	200 GB/s	300 GB/s
On-chip memory	1 MB	0.5 MB	6 MB	11 MB
Maximum Power	120W	108 W	-	-
Double Precision	NO	YES	YES	YES
Price	350 $	900 $	3000 $	37000 $

Even though companies like Microsoft may not disclose their data-center infrastructure specification in detail, reports suggest that a typical data center can consume about 30 MW and include about 50,000 servers, with one or two GPUs on each card. For example, the Microsoft Azure cloud service offers Tesla K80 cards with two GK 210 GPUs on each card,as illustrated in figure 4.14. A Tesla GK 210 has a similar specification to our K4200 GPU in terms of core frequency (562MHz), architecture(Kepler) and double precision support.

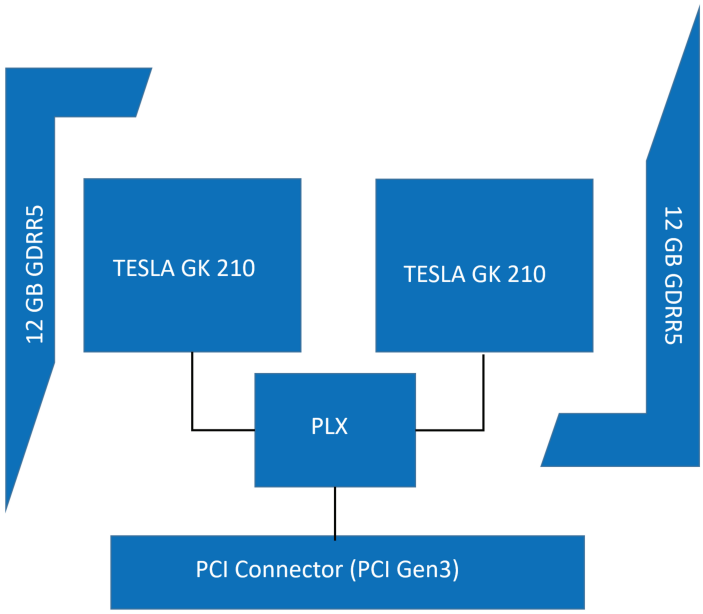


Figure 4.14 Tesla K80 Block Diagram

Figures 4.15 and 4.16 compare the performance of two discussed test cases on target GPUs and FPGAs, with increasing input table sizes. For used benchmarks, the most advanced VU440 FPGA has better performance than both GPUs. In this experiment, our FPGA implementation uses a 200 MHz clock frequency that results in lower dynamic power consumption and better overall performance per watt (i.e. better energy consumption) than both GPU platforms. Tables 4.5 and 4.6 present performance, resource usage and power analysis for the two discussed algorithms, using always the same data size (8192 items[2]). In the FPGA case, we instantiated a number of WGs that uses at most about 60% of the on-chip resources, to ensure that the design can be placed and routed[3].

[2]8192 elements need 256 kB memory which is the size of available L1 cache for K4200 GPU

[3]The current version of SDAccel from Xilinx also limits the maximum number of WGs that can be instantiated on an FPGA to 10. We did not consider this limitation since it is tool-dependent, rather than resource-dependent, and will most likely be lifted in future versions of the tool.

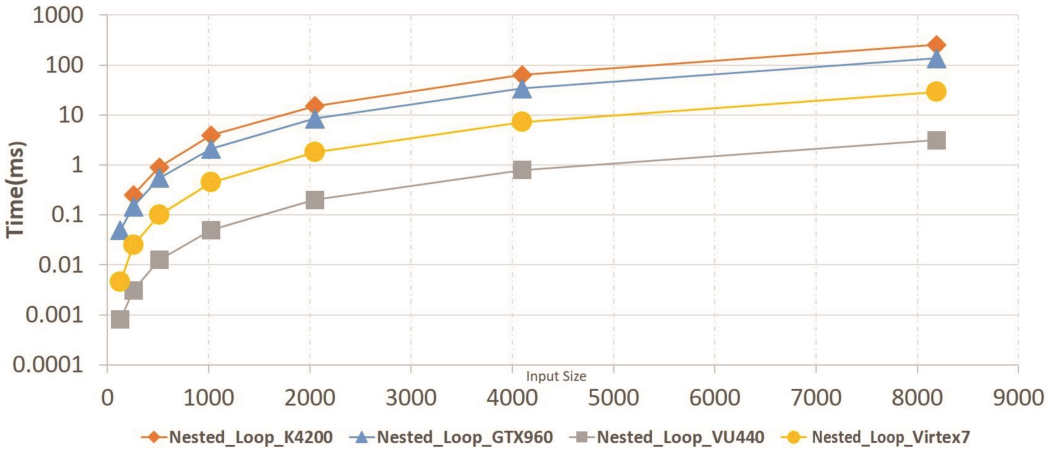


Figure 4.15 Performance comparison of nested-loop join versus data size

Moreover, for both applications a fully-optimized implementation on both FPGAs consumes less energy than tested GPUs to perform the same amount of computation. This is due to the smaller power consumption, and in case of the VU440 also to a lower execution time.

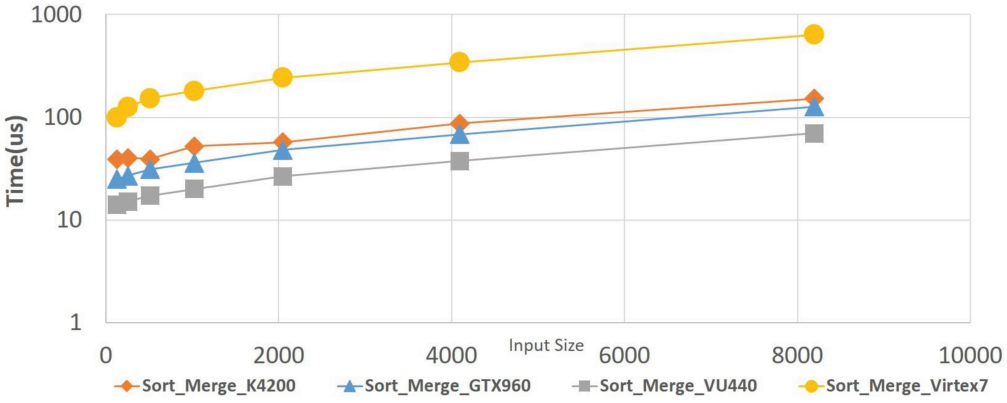


Figure 4.16 Performance comparison of sort-merge join versus data size

Table 4.5 Performance and energy analysis of Nested_Loop Join

Params/Devices	**Virtex7**	**VU440**	**GTX960**	**K4200**
Device time	29 ms	3.145 ms($t_{clk} = 5ns$)	135 ms	253ms
WGs	65	400	256	256
Band-Width Utilization	3.2 %(6.5 GB/s)	20 %(60 GB/s)	100% (172 GB/s)	59%$(66GB/s)$
Device power	35 W	81.7 W	95 W	105 W
Energy	1 J	0.256 J	12.8 J	26.5 J
Utilization	BRAMs = 65(4.4%)	400(16%)	NA	
	DSPs = 260(7.2%)	1600(28%)		
	FFs= 279500(32%)	1720000(33%)		
	LUTs = 260000(60%)	1600000(63%)		

Table 4.6 Performance and energy analysis of Sort_Merge Join

Params/Devices	**Virtex7**	**VU440**	**GTX960**	**K4200**
Device time	638 us	70 us ($t_{clk} = 5ns$)	127 us	152 us
WGs	20	122	256	256
Band-Width Utilization	5% (10 GB/s)	30% (90 GB/s)	100 % (172 GB/s)	42% (47 GB/s)
Device power	13.2 W	75 W	90 W	100 W
Energy	8.4 mJ	5.2 mJ	11.7 mJ	15.2 mJ
Utilization	BRAMs = 140 (9.5 %)	923 (37%)	NA	
	DSPs = 300(8.3%)	2023 (36%)		
	FFs = 268000(30%)	1634800 (32%)		
	LUTs = 254800 (58%)	1554400 (61%)		

Fast and extensive design space exploration of generated RTL is paramount to achieve most optimum solution based on power and execution time. Appendix A describes a developed model in MATLAB which evaluates all solutions with respect to default behavior of SDAccel considering area , performance and dynamic power consumption of each solution and suggests the best solution with highest overall performance-per-watt.

4.2.4 FPGAs and Energy Saving

A trade-off between power consumption and performance is crucial for designing high performance accelerators to avoid reaching power budget limit. Thoughtful utilization of power budget depends on both design architecture and underlying hardware. Figure 4.17 draws a comparison between GPU and FPGA devices in terms of performance-per-watt of each target device running join operation.

Virtex7 device performs same amount of computation as K4200 and GTX960 while consuming less power, on the other hand, VU440 is significantly more power efficient and can outperform high-end GPUs. VU440 (20 nm) shows 55 % higher performance-per-watt in comparison to GTX960 with MAXWELL architecture, moreover, Virtex7 device is 21% more energy-efficient with respect to GTX960 manufactured in the same node as tested GPUs (28 nm).

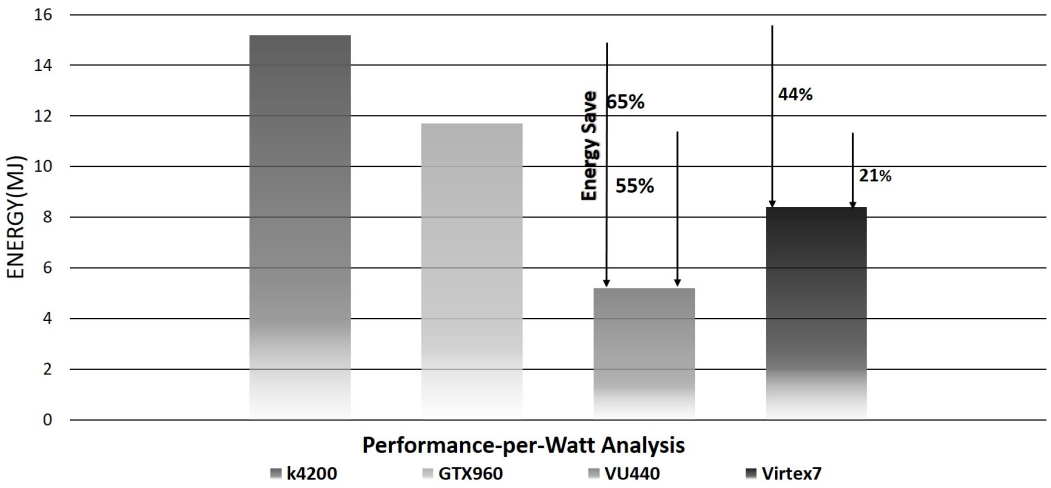


Figure 4.17 Performance-per-watt compassion of FPGA vs GPU

All in all, in this section we compared performance and energy consumption of two well-known join algorithm implementations on GPU and FPGA devices. Nested-loop and sort-merge join algorithms are memory-intensive computations that require careful optimization to be efficiently implemented on FPGAs. Our experiment suggests that a significant amount of speed up can be achieved by properly using all the optimization techniques offered by SDAccel. Note that, even though sorting is a memory-intensive application, our best implementation makes such effective use of the available DRAM bandwidth that it has better performance than a GPU. Moreover, thanks to the lower power consumption of the FPGA, its overall energy consumption per operation is significantly better than that of a GPU.

Chapter 5

Conclusion

In this dissertation, several aspects of high performance computing (HPC) were discussed. In chapter one and three, GPUs and FPGAs architecture and technology advancements are studied to provide important insights into HPC's underlying hardware.

In chapter two, high level synthesis design process is introduced and discussed in detail, Xilinx high-level synthesis tool chain is used in this research to study and exercise FPGA programming using high level machine languages e.g. C/C++, OpenCL and System C. Additionally, it is studied how compilation flow of C based model toward high quality RTL is possible through HLS design flow with minimum required hardware knowledge using wide range of optimization techniques provided by modern HLS tool.

With the aim of computation acceleration, two different hardware design approaches are studied in this PhD program. Firstly, Model Based Design(MBD) is used to develop digital signal processing algorithm of radar application discussed in chapter 3. To do so, Simulink (developed by MATLAB) is used to model the algorithm and generate C/C++ and HDL code from top model to compare high-level and RTL synthesis using Simulink IPs. Secondly, as an alternative approach to MBD, OpenCL programming language is used to program and compare GPU and FPGA devices using standard OpenCL benchmarks via SDAccel which is the Xilinx SDK for OpenCL. The experimental results suggest that MBD approach enables designers to perform hardware-software co-design using verified sub-systems with small design space which are only able to address limited range of applications, on the other hand,

OpenCL framework with stronger design space exploration strategies and portable functionality can program CPU/GPU and recently FPGA devices. OpenCL framework enables developers to accelerate wide range of workloads coming from various domains through more laborious design cycles with respect to MBD methodology.

In chapter three, various performance hungry applications are algorithmically analyzed to shed lights on hardware acceleration challenges and requirements. Moreover, techniques and principles of off-chip bandwidth utilization, on-chip pipelining and parallelism which are paramount to design high performance accelerators are covered in this work.

Several options offered by the SDAccel tool from Xilinx were utilized to optimize the application code for FPGA implementation. They mainly included pipelining work-items and using on-chip global memory buffers for inter-kernel communications rather than using the traditional slower off-chip DRAM based global memory buffers. Burst memory accesses were used for accessing the off-chip global memory resulting in higher efficiency, since the access over-head is shared between larger amounts of data being transferred. Concurrency on the FPGA was exploited further by splitting the overall kernel computations into smaller chunks and executing them in parallel using multiple compute units. All these optimizations were complemented by the conventional HLS-based data-path optimization options e.g. pipelining and unrolling both the explicit and the implicit loops in the kernels (i.e. the loops over work-items).

Finally, this work reports how design space of multi-core RTL is explored using different verified solutions via proposed Matlab model discussed in appendix A that enables a designer to choose most optimum solution based on execution time, power consumption and area of each application. This approach proposes automated flow to evaluate many solutions each with unique optimization setting based on the formulated criteria to design ASIC-like hardware from high-level model. The main motivation behind developing this model relies on the wide range of optimization options provided by HLS tools which demands careful exploration of design space to choose proper optimization for each application. Although, our experiments suggest GPU outperforms FPGA, but FPGA has higher performance-per-watt in comparison to GPUs manufactured in the same node(28 nm). Additionally, our experiments report that 20 nm Xilinx FPGAs (Xilinx UltraScale) with much larger amount of

logic and on-chip memory consume less power and outperform NVIDIA GPUs manufactured with 28 nm transistors at the cost of more expensive devices[1].

We intend to exploit the findings from this research activity in the future to enhance the level of automation in existing HLS tools to obtain high performance energy efficient acceleration of kernels written in non hardware-specific OpenCL by using FPGA-based platforms.

[1]NVIDIA GPUs and Xilinx FPGAs are discussed in chapter three

Bibliography

[1] Altera. Quartus ® prime standard edition handbook volume 1: Design and synthesis, 2017.

[2] Xilinx. Vivado design suite user guide:ug893, 2016.

[3] Peter Kevin. *Xilinx vs Intel "The New Showdown in Programmable Logic"*. Electronic Engineering Journal, April,2016.

[4] Tanner paige. *Could Xilinx Benefit from the Technology Industry's Shift to IoT?* Electronic Engineering Journal, July,2017.

[5] *Field Programmable Gate Array (FPGA) Market Size By Application*. Gloabl Market Insights, February,2016.

[6] *Field Programmable Gate Array (FPGA) Market Analysis By Technology (SRAM, EEPROM, Antifuse, Flash), By Application (Consumer Electronics, Automotive, Industrial, Data Processing, Military & Aerospace, Telecom), And Segment Forecasts, 2014 - 2024*. GRAND VIEW RESEARCH, December,2016.

[7] Jeff Dorsch. *A Chip For All Seasons*. Semiconductor Engineering, September,2017.

[8] Mohammad Al-Fares, Sivasankar Radhakrishnan, Barath Raghavan, Nelson Huang, and Amin Vahdat. Hedera: Dynamic flow scheduling for data center networks. In *NSDI*, volume 10, pages 19–19, 2010.

[9] Integrating 100-gbe switching solutions on 28-nm fpgas. In *NSDI*, volume 10, 2010.

[10] *7 Series FPGAs Configurable Logic Block*. Xilinx, September,2016.

[11] *7 Series DSP48E1 Slice, UG479 (v1.9)*. Xilinx, September,2016.

[12] *7 Series FPGAs Memory Resources*. Xilinx, September,2016.

[13] Madhu Monga Martin S.Won. Intel® arria® 10 fpga performance benchmarking methodology and results. In *White Paper*. Intel, 2017.

[14] FPGA Intel. Sdk for opencl. *Programming Guide. UG-OCL002*, 31, 2016.

[15] Shahzad Ahmad Butt, Mehdi Roozmeh, and Luciano Lavagno. Designing parameterizable hardware ips in a model-based design environment for high-level synthesis. *ACM Transactions on Embedded Computing Systems (TECS)*, 15(2):32, 2016.

[16] Mehdi Roozmeh Luciano Lavagno. Implementation of a performance optimized database join operation on fpga-gpu platforms using opencl. In *NORCHIP and International Symposium of System-on-Chip (SoC)*. IEEE, 2017.

[17] Liang; Roozmeh Mehdi; Lavagno Luciano Muslim, Fahad Bin; Ma. *Efficient FPGA Implementation of OpenCL High-Performance Computing Applications via High-Level Synthesis*. IEEE ACCESS, 2017.

[18] Zheming Jin, Hal Finkel, Kazutomo Yoshii, and Franck Cappello. Evaluation of the fir example using xilinx vivado high-level synthesis compiler. Technical report, Argonne National Lab.(ANL), Argonne, IL (United States), 2017.

[19] Colin Yu Lin, Zhenghong Jiang, Cheng Fu, Hayden Kwok-Hay So, and Haigang Yang. Fpga high-level synthesis versus overlay: Comparisons on computation kernels. *ACM SIGARCH Computer Architecture News*, 44(4):92–97, 2017.

[20] Timothy Prickett Morgan. *Intel Gears Up For FPGA Push*. The Next Platfrom, October 2, 2017.

[21] Xilinx. Vivado design suite user guide- power analysis and optimization - ug907. page 104, 2015.

[22] *The Xilinx SDAccel Development Environment "Bringing The Best Performance/Watt to the Data Center"*. Xilinx, 2014.

[23] Sdaccel environment optimization guide. In *UG1207*, page 38. Xilinx, August-2016.

[24] *Vivado Design Suite User Guide*. Xilinx, 2014.

[25] Sdaccel development environment help. In *UG1188*. Xilinx, Ferbruary-2018.

[26] Dana Schaa Dong Ping Zhang David R.kaeli, Perhaad Mistry. *Heterogeneous Computing with OpenCL 2.0*. Xilinx, 2014.

[27] Lester Kalms and Diana Göhringer. Exploration of opencl for fpgas using sdaccel and comparison to gpus and multicore cpus. In *Field Programmable Logic and Applications (FPL), 2017 27th International Conference on*, pages 1–4. IEEE, 2017.

[28] Developer Board for Acceleration with KU115. Platform reference design user guide, ug1234. Xilinx, 2016.

[29] Chao Li, Yanjing Bi, Yannick Benezeth, Dominique Ginhac, and Fan Yang. High-level synthesis for fpgas: code optimization strategies for real-time image processing. *Journal of Real-Time Image Processing*, pages 1–12, 2017.

[30] Razvan Nane, Vlad-Mihai Sima, Christian Pilato, Jongsok Choi, Blair Fort, Andrew Canis, Yu Ting Chen, Hsuan Hsiao, Stephen Brown, Fabrizio Ferrandi, et al. A survey and evaluation of fpga high-level synthesis tools. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 35(10):1591–1604, 2016.

[31] Aydin Emre Guzel, Vecdi Emre Levent, Mustafa Tosun, M Akif Özkan, Toygar Akgun, Duygu Büyükaydin, Cengiz Erbas, and H Fatih Ugurdag. Using high-level synthesis for rapid design of video processing pipes. In *East-West Design & Test Symposium (EWDTS), 2016 IEEE*, pages 1–4. IEEE, 2016.

[32] Yakun Sophia Shao, Brandon Reagen, Gu-Yeon Wei, and David Brooks. Aladdin: A pre-rtl, power-performance accelerator simulator enabling large design space exploration of customized architectures. In *Computer Architecture (ISCA), 2014 ACM/IEEE 41st International Symposium on*, pages 97–108. IEEE, 2014.

[33] Alessandro Cilardo and Luca Gallo. Interplay of loop unrolling and multidimensional memory partitioning in hls. In *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2015*, pages 163–168. IEEE, 2015.

[34] Anirban Sengupta, Saumya Bhadauria, and Saraju P Mohanty. Tl-hls: Methodology for low cost hardware trojan security aware scheduling with optimal loop unrolling factor during high level synthesis. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 36(4):655–668, 2017.

[35] Benjamin Carrion Schafer. Enabling high-level synthesis resource sharing design space exploration in fpgas through automatic internal bitwidth adjustments. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 36(1):97–105, 2017.

[36] Benjamin Carrion Schafer. Parallel high-level synthesis design space exploration for behavioral ips of exact latencies. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 22(4):65, 2017.

[37] Shahzad Ahmad Butt and Luciano Lavagno. Design space exploration and synthesis for digital signal processing algorithms from simulink models. In *Design and Test Symposium (IDT), 2013 8th International*, pages 1–6. IEEE, 2013.

[38] Shahzad Ahmad Butt, Parinaz Sayyah, and Luciano Lavagno. Model-based hardware/software synthesis for wireless sensor network applications. In *Electronics, Communications and Photonics Conference (SIECPC), 2011 Saudi International*, pages 1–6. IEEE, 2011.

[39] Hung-Yi Liu and Luca P Carloni. On learning-based methods for design-space exploration with high-level synthesis. In *Design Automation Conference (DAC), 2013 50th ACM/EDAC/IEEE*, pages 1–7. IEEE, 2013.

[40] Mohammad Motamedi, Philipp Gysel, Venkatesh Akella, and Soheil Ghiasi. Design space exploration of fpga-based deep convolutional neural networks. In *Design Automation Conference (ASP-DAC), 2016 21st Asia and South Pacific*, pages 575–580. IEEE, 2016.

[41] Whitepaper. *NVIDIA'S Next Generation CUDA Comute Architecture : Kepler GL110/210*. NVIDIA, 2014.

[42] Jan Vanek, Josef Michalek, and Josef Psutka. A gpu-architecture optimized hierarchical decomposition algorithm for support vector machine training. *IEEE Transactions on Parallel and Distributed Systems*, 2017.

[43] Siva Kumar Sastry Hari, Timothy Tsai, Mark Stephenson, Stephen W Keckler, and Joel Emer. Sassifi: An architecture-level fault injection tool for gpu application resilience evaluation. In *Performance Analysis of Systems and Software (ISPASS), 2017 IEEE International Symposium on*, pages 249–258. IEEE, 2017.

[44] Sparsh Mittal. *A Survey of Techniques for Architecting and Managing GPU Register File*. IEEE TRANSACTIONS ON PARALLEL AND DISTRIBUTING SYSTEMS, 2017.

[45] Mark Harris. *Maxwell: The Most Advanced CUDA GPU Ever Made*. NVIDIA, 2014.

[46] High bandwidth memory (hbm) dram. 2015.

[47] John H Lau. Overview and outlook of three-dimensional integrated circuit packaging, three-dimensional si integration, and three-dimensional integrated circuit integration. *Journal of Electronic Packaging*, 136(4):040801, 2014.

[48] Mingxian Chen and Zhi Zhong. Block nested join and sort merge join algorithms: An empirical evaluation. In *International Conference on Advanced Data Mining and Applications*, pages 705–715. Springer, 2014.

[49] Cagri Balkesen, Gustavo Alonso, Jens Teubner, and M Tamer Özsu. Multi-core, main-memory joins: Sort vs. hash revisited. *Proceedings of the VLDB Endowment*, 7(1):85–96, 2013.

[50] Ren Chen, Sruja Siriyal, and Viktor Prasanna. Energy and memory efficient mapping of bitonic sorting on fpga. In *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 240–249. ACM, 2015.

[51] Rene Mueller, Jens Teubner, and Gustavo Alonso. Sorting networks on fpgas. *The VLDB Journal—The International Journal on Very Large Data Bases*, 21(1):1–23, 2012.

[52] Qi Mu, Liqing Cui, and Yufei Song. The implementation and optimization of bitonic sort algorithm based on cuda. *arXiv preprint arXiv:1506.01446*, 2015.

[53] Peter A Boncz, Marcin Zukowski, and Niels Nes. Monetdb/x100: Hyper-pipelining query execution. In *Cidr*, volume 5, pages 225–237, 2005.

[54] Zeke Wang, Johns Paul, Hui Yan Cheah, Bingsheng He, and Wei Zhang. Relational query processing on opencl-based fpgas. In *Field Programmable Logic and Applications (FPL), 2016 26th International Conference on*, pages 1–10. IEEE, 2016.

[55] Mohamed S Abdelfattah, Andrei Hagiescu, and Deshanand Singh. Gzip on a chip: High performance lossless data compression on fpgas using opencl. In *Proceedings of the International Workshop on OpenCL 2013 & 2014*, page 4. ACM, 2014.

[56] Konstantinos Krommydas, Ruchira Sasanka, and Wu-chun Feng. Bridging the fpga programmability-portability gap via automatic opencl code generation and tuning. In *Application-specific Systems, Architectures and Processors (ASAP), 2016 IEEE 27th International Conference on*, pages 213–218. IEEE, 2016.

[57] Dsp builder for intel fpgas introduction. Intel, 2017.

[58] Altera. Implementing digital signal processing for automotive radar using socs. Altera, 2013.

[59] Nvidia simulation. *https://developer.download.nvidia.com/compute/DCGM/docs/nvidia-smi-367.38.pdf*, page 34.

[60] David Barrie Thomas, Lee Howes, and Wayne Luk. A comparison of cpus, gpus, fpgas, and massively parallel processor arrays for random number generation. In *Proceedings of the ACM/SIGDA international symposium on Field programmable gate arrays*, pages 63–72. ACM, 2009.

[61] Fahad Bin Muslim, Liang Ma, Mehdi Roozmeh, and Luciano Lavagno. Efficient FPGA implementation of opencl high-performance computing applications via high-level synthesis. *IEEE Access*, 5:2747–2762, 2017.

Appendix A

The Matlab code reported in the next page, evaluates all customized solutions based on three discussed criteria in chapter two and propose the best solution with highest value of performance-per- watt. Obviously, model can be improved by considering wider set of results and more numbers. This model was used to obtain presented results in chapter 4 and in [16, 61].

Table of Contents

VARIABLES and CONSTANT DECLARATION

```
solution=12; % Number of solutions that you have provided using
 SDAccel
Time_constraint=1000; %Time Constraints of the design for specified
 data size in microsecond (array length=8192)

%%%%%% Following variables stores characteristics of the used device
 during

%%%%%% SDAccel simulation (KU115 - KINTEX ultra scale)
BANDWIDTH_DEFAULT=10;  %% DDR memory bandwidth(GB/s) of the used
 device during simulation
LUT_DEFAULT=663600;     %% Available LUT of the used device during
 simulation
FF_DEFAULT=1326720;     %% Available FF of the used device during
 simulation
DSP_DEFAULT=5520;       %% Available DSP of the used device during
 simulation
BRAM_DEFAULT=2160;      %% Available BRAM of the used device during
 simulation

%%%%%Following variables stores characteristics of target platform
 (XCVU440 - VIRTEX ULTRASCALE)

BANDWIDTH_TARGET=20;  %% DDR memory bandwidth(GB/s) of the target
 device
LUT_TARGET=2532960;     %% Available LUT of the target device

FF_TARGET=5065920;      %% Available FF of the target device
DSP_TARGET=5520;        %% Available BRAM of the target device
BRAM_TARGET=2520;       %% Available DSP of the target device


%%%%% Following coefficients are used in order to estimate
 characteristics of
%%%%% the design on the target platform. SDAccel does not provide
 users with all available Xilinx FPGAs and it considers same bandwidth
 for all available devices(~ 10 GB/s).
Bandwidth_CO=BANDWIDTH_TARGET/ BANDWIDTH_DEFAULT;
LUT_CO= LUT_TARGET/ LUT_DEFAULT;
FF_CO = FF_TARGET / FF_DEFAULT;
```

```
BRAM_CO =BRAM_TARGET / BRAM_DEFAULT;
DSP_CO = DSP_TARGET / DSP_DEFAULT ;

%PREALLOCATION OF VARIBALES THAT STORES PARAMETERS FROM SDACCEL FOR
 EACH
%KERNEL USING DIFFERENT SOLUTIONS ON KU115 DEVICE, EXECUTION TIME IS
%OBTAINED FROM HARDWAARE EMULATION (SINGLE COMPUTE) AND RESOURCE
 UTILIZATION REPORT GENERATED BY
%VIVADO DESIGN SUIT.

time_sortlocal=zeros(solution ,1);
time_mergelocal=zeros(solution ,1);
time_mergeglobal=zeros(solution ,1);

LUT_sortlocal=zeros(solution,1);
LUT_mergelocal=zeros(solution,1);
LUT_mergeglobal=zeros(solution,1);
FF_sortlocal=zeros(solution,1);
FF_mergelocal=zeros(solution,1);
FF_mergeglobal=zeros(solution,1);
BRAM_sortlocal=zeros(solution,1);
BRAM_mergelocal=zeros(solution,1);
BRAM_mergeglobal=zeros(solution,1);
DSP_sortlocal=zeros(solution,1);
DSP_mergelocal=zeros(solution,1);
DSP_mergeglobal=zeros(solution,1);
%%%%DYNAMIC POWER CONSUMPTION OF SINGLE COMPUTE UNIT(W)
Power_sortlocal=zeros(solution,1);
Power_mergelocal=zeros(solution,1);
Power_mergeglobal=zeros(solution,1);
%%%STATIC POWER
P_STATIC=1.25;
%%%%%%%%%%
call_sortlocal=zeros(solution,1);   %% CALL NUMBER OF THE SORTLOCAL
 COMPUTE UNIT IN ORDER TO COMPLETE COMPUTATION
call_mergelocal=zeros(solution,1);  %% CALL NUMBER OF THE MERGELOCAL
 COMPUTE UNIT IN ORDER TO COMPLETE COMPUTATION
call_mergeglobal=zeros(solution,1); %% CALL NUMBER OF THE MERGEGLOBAL
 COMPUTE UNIT IN ORDER TO COMPLETE COMPUTATION
quality_sortlocal=zeros(solution,1); % Quality of sortlocal kernel
 considering LUT , exectution time and number of call
quality_mergelocal=zeros(solution,1); % Quality of mergelocal kernel
 considering LUT , exectution time and number of call
quality_mergeglobal=zeros(solution,1); % Quality of mergeglobal kernel
 considering LUT , execution time and number of call
Time_vector = zeros(solution,1) ;
Call_vector = zeros(solution,1) ;
quality_matrix=zeros(solution , 3);
Time_matrix=zeros(solution , 3);
Call_matrix=zeros(solution , 3);

%%%%%%%READ DATA FROM EXCEL FILE AND STORE EACH COLUMN IN A ARRAY
data=xlsread('sheet3.xlsx'); % Read the data from excel file
```

```
LUT=data(:,1);
FF=data(:,2);
BRAM=data(:,3);
DSP=data(:,4);
time=data(:,5);
call=data(:,6);
local=data(:, 7);
Power=data(:, 10);


%%%% THIS LOOP GENERATES UNIQUE ARRAY FOR EACH KERNEL THAT STORE
 EXECUTION
%%%% TIME, RESOURCE UTILIZATION (PERCENTAGE) AND DYNAMIC POWER
 CONSUMPTION.

for i= 0 : solution-1

time_sortlocal(i+1 , :) = time( 3*i + 2  , :);
time_mergelocal(i+1 , :) = time( 3*i + 3  , :);
time_mergeglobal(i+1 , :) = time( 3*i + 4  , :);
LUT_sortlocal(i+1 , :) = LUT(3*i + 2   , :);
LUT_mergelocal(i+1 , :) = LUT(3*i + 3   , :);
LUT_mergeglobal(i+1 , :) = LUT(3*i + 4   , :);
FF_sortlocal(i+1 , :) = FF(3*i + 2   , :);
FF_mergelocal(i+1 , :) = FF(3*i + 3   , :);
FF_mergeglobal(i+1 , :) = FF(3*i + 4   , :);
BRAM_sortlocal(i+1 , :) = BRAM(3*i + 2   , :);
BRAM_mergelocal(i+1 , :) = BRAM(3*i + 3   , :);
BRAM_mergeglobal(i+1 , :) = BRAM(3*i + 4   , :);
DSP_sortlocal(i+1 , :) = DSP(3*i + 2   , :);
DSP_mergelocal(i+1 , :) = DSP(3*i + 3   , :);
DSP_mergeglobal(i+1 , :) = DSP(3*i + 4   , :);
call_sortlocal(i+1 , :) = call(3*i + 2   , :) ;
call_mergelocal(i+1 , :) = call(3*i + 3   , :) ;
call_mergeglobal(i+1 , :) = call(3*i + 4   , :) ;
Power_sortlocal(i+1 , :) = Power(3*i + 2   , :) ;
Power_mergelocal(i+1 , :) = Power(3*i + 3   , :) ;
Power_mergeglobal (i+1 , :) = Power(3*i + 4   , :) ;

end
%%%%% Total computation time of each solution before quality
 assessment

for i=0 : solution-1
    Time_matrix( i+1 , 1  )  = call_sortlocal(i+1 , :) *
 time_sortlocal(i+1 , :);
    Time_matrix( i+1 , 2  )  = call_mergelocal(i+1 , :) *
 time_mergelocal(i+1 , :);
    Time_matrix( i+1 , 3  )  = call_mergeglobal(i+1 , :)*
 time_mergeglobal(i+1 , :);
    Call_matrix( i+1 , 1  )  = call_sortlocal(i+1 , :);
    Call_matrix( i+1 , 2  )  = call_mergelocal(i+1 , :) ;
    Call_matrix( i+1 , 3  )  = call_mergeglobal(i+1 , :) ;
end
```

```
for i=0 : solution-1
   Time_vector(i+1 , :) = Time_matrix(i+1 ,1)+ Time_matrix(i+1, 2)+
 Time_matrix(i+1, 3);
   Call_vector(i+1 , :) = call_sortlocal(i+1 , :)+call_mergelocal(i
+1 , :)+call_mergeglobal(i+1 , :);
end
%%%%%%%%%%%
```

SOLUTION ASSESSMENTS

EVALUATION OF EACH SOLUTION CONSIDERING AREA AND EXECUTION TIME

```
for i=0 : solution-1

   quality_sortlocal(i+1,:)= (time_sortlocal(1 , :) /
 time_sortlocal(i+1,:))*(LUT_sortlocal(1, :)/LUT_sortlocal(i
+1,:))*(call_sortlocal(1,:)/ call_sortlocal(i+1,:));

   quality_mergelocal(i+1,:)= (time_mergelocal(1 , :) /
 time_mergelocal(i+1,:))*(LUT_mergelocal(1, :)/LUT_mergelocal(i
+1,:))*(call_mergelocal(1,:)/ call_mergelocal(i+1,:));

   quality_mergeglobal(i+1,:)= (time_mergeglobal(1 , :) /
 time_mergeglobal(i+1,:))*(LUT_mergeglobal(1, :)/LUT_mergeglobal(i
+1,:))*(call_mergeglobal(1,:)/ call_mergeglobal(i+1,:));

end
```

```
[M_local,I_local] = max(quality_sortlocal(:));      % CHOOSING BEST
 SULTION WITH HIGHEST QUALITY FACTOR
[M_global,I_global] = max(quality_mergeglobal(:));  % CHOOSING BEST
 SULTION WITH HIGHEST QUALITY FACTOR

 for i=0 : solution-1


            quality_matrix(i+1 , 1) = quality_mergeglobal(i+1 , :);
            quality_matrix(i+1 , 2) = quality_sortlocal(i+1 , :);
            quality_matrix(i+1 , 3) = quality_mergelocal(i+1 , :);

     end
```

PERFORMANCE ANALYSIS

TOTAL EXECUTION TIME OF WHOLE ALGORITHM USING KERNELS WITH HIGHEST QUALITY

```
Time = (call_sortlocal(I_local,: )* time_sortlocal(I_local, :)
 +call_mergelocal(I_global,: )*
```

```
 time_mergelocal(I_global, :)+call_mergeglobal(I_global,: )*
 time_mergeglobal(I_global, :)) /Bandwidth_CO ;
```

NUMBER OF COMPUTE UNITS

```
compute_unit = ceil( Time/ Time_constraint);

%%%% EXECUTION TIME OF WHOLE ALGORITHM AFTER INSTANTIATION OF MULTIPLE
 COMPUTE UNITS

Time_Final= Time / compute_unit;  %%% EXECUTION TIME OF ALGORITHM
 USING MULTIPLE COMPUTE UNIT
```

RESOURCE UTILIZATION

RESOURCE UTILIZATION OF THE DESIGN AFTER INSTANTIATION OF MULTIPLE COMPUTE UNIT ON TARGET PLATFORM (XCVU440 - VIRTEX ULTRASCALE)

```
TOTAL_LUT_PERCENTAGE = (LUT_sortlocal(I_local, :)* compute_unit +
 LUT_mergelocal(I_global)* compute_unit+LUT_mergeglobal(I_global)*
 compute_unit) / LUT_CO;
TOTAL_FF_PERCENTAGE = (FF_sortlocal(I_local, :)* compute_unit +
 FF_mergelocal(I_global)* compute_unit+FF_mergeglobal(I_global)*
 compute_unit) / FF_CO;
TOTAL_BRAM_PERCENTAGE = (BRAM_sortlocal(I_local, :)* compute_unit +
 BRAM_mergelocal(I_global)* compute_unit+BRAM_mergeglobal(I_global)*
 compute_unit) / BRAM_CO;
TOTAL_DSP_PERCENTAGE = (DSP_sortlocal(I_local, :)* compute_unit +
 DSP_mergelocal(I_global)* compute_unit+DSP_mergeglobal(I_global)*
 compute_unit) / DSP_CO;

TOTAL_LUT_NUMBER=  (TOTAL_LUT_PERCENTAGE * LUT_TARGET)/100;
TOTAL_FF_NUMBER = (TOTAL_FF_PERCENTAGE *FF_TARGET)/100;
TOTAL_BRAM_NUMBER= (TOTAL_BRAM_PERCENTAGE * BRAM_TARGET)/100;
TOTAL_DSP_NUMBER = (TOTAL_DSP_PERCENTAGE * DSP_TARGET)/100;
```

POWER CONSUMPTION

TOTAL ON-CHIP POWER USING MULTIPLE COMPOUTE UNITS

```
TOTAL_POWER=(Power_sortlocal(I_local, :)* compute_unit +
 Power_mergelocal(I_global)* compute_unit+Power_mergeglobal(I_global)*
 compute_unit) + P_STATIC ;

%bar3(quality_matrix)
%bar3(Time_matrix)
```

Published with MATLAB® R2017a